srcQL: A Syntax-Aware Query Language for Source Code

Brian Bartman
Computer Science
Kent State University
Kent, Ohio, USA
bbartman@cs.kent.edu

Christian D. Newman
Computer Science
Kent State University
Kent, Ohio, USA
cnewman@kent.edu

Michael L. Collard
Computer Science
The University of Akron
Akron, Ohio, USA
collard@uakron.edu

Jonathan I. Maletic
Computer Science
Kent State University
Kent, Ohio, USA
jmaletic@cs.kent.edu

***Abstract*—A tool and domain specific language for querying source code is introduced and demonstrated. The tool, srcQL, allows for the querying of source code using the syntax of the language to identify patterns within source code documents. srcQL is built upon srcML, a widely used XML representation of source code, to identify the syntactic contexts being queried. srcML inserts XML tags into the source code to mark syntactic constructs. srcQL uses a combination of XPath on srcML, regular expressions, and syntactic patterns within a query. The syntactic patterns are snippets of source code that supports the use of logical variables which are unified during the query process. This allows for very complex patterns to be easily formulated and queried. The tool is implemented (in C++) and a number of queries are presented to demonstrate the approach. srcQL currently supports C++ and scales to large systems.**

Keywords—Source code querying, syntactic search, srcML

I. Introduction

Currently, search tools in IDEs (Integrated Development Environment) typically use regular-expression matching. The main drawback with regular expression or text based search tools is that too many, non-relevant, parts of source code are matched. This leads to developers manually filtering the results (sometimes 1000's of matches). While these matching techniques are very powerful, they are intended to be applied across *any* text-based document.

The research presented here takes a different viewpoint. Since developers are searching and exploring source code, we feel that the search and matching tools will benefit greatly from being inherently aware of the underlying syntax of the programming language. We present our implementation of a syntax-aware query language that is built on top of a syntax-aware pattern-matching language. The query language is called srcQL (source code Query Language) and the pattern matching language is called srcPat (source code Pattern matching).

The objective is to give developers the ability to easily construct queries that take into account the syntactic features in the programming language. For example, a developer may want to find all functions in a system that have a call to both *open* and *close* within the body. Furthermore, the ordering may be of interest; *open* before *close*. This can be further refined to locating all calls to *open* within a guard. That is, find all calls to *open* within a body of an if-statement. These types of queries are very difficult, if not impossible, to describe as regular expressions.

There are a number of syntactic pattern matching approaches, typically used by transformation languages [5][6]. While these provide some support for syntax aware querying, they are fairly complicated to use and/or require a deep understanding of the underlying language grammar. They also typically work on the abstract syntax tree, as generated by the compiler, and thus do not match one-to-one with the actual source code documents. This tree-based matching is also fairly inefficient and as such cannot be practically applied in real time within the context of an IDE. Because of these various issues none are widely adopted by developers.

To overcome some of these roadblocks we developed srcQL. srcQL is built on top of the srcML infrastructure [1] which allows us to leverage XML technologies (i.e., XPath). srcML is a widely used, highly scalable, parsing technology and intermediate representation that marks up source code with abstract syntactic information in the form of XML tags. srcQL queries are compiled into XPath and then applied to the srcML representation of the source code. This addresses, to a large extent, the scalability and efficiency problems. Moreover, we modeled srcQL after the relational query language SQL. This makes developing and understanding the meaning of srcQL queries fairly natural for developers (at least those familiar with SQL).

The design and implementation of this syntax-aware query language for C++ is presented and demonstrated. The main contributions of this tool include: 1) Syntactic pattern matching and querying with unification; 2) Support for query relations of containment, partial ordering, functional constraints, and syntactic information.

The paper is organized as follows. The next section describes the overall architecture of the srcQL language and the underlying infrastructure used. Section III describes srcPat and section IV presents the details of srcQL. A demonstration of using srcQL is presented in section 0. This is followed by related work and conclusions.

II. srcQL Architecture

We take an open layered approach in architecting srcQL. **Figure 1** presents the main components. srcML [1-3] is the base layer and provides the syntactic information. In short, srcML is an XML format used to augment source code with syntactic information from the AST to add explicit structure to program source code. srcPat expressions are converted into XPath,

which is then applied to srcML documents (i.e., source code with XML markup).

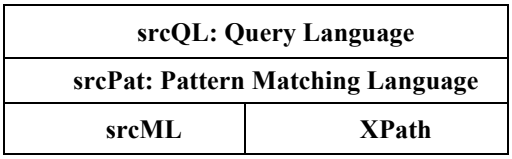

Figure 1. srcQL is built on top of pattern matching language, srcPat. This in turn is built on top of srcML and XPath.

The underlying infrastructure for srcQL is the srcML format and its associated toolkit (see www.srcML.org). It allows source code to be converted to the srcML format and then back to source code with no loss of any original textual information.

The approach is also based on XPath, which is a standardized way of querying XML documents in a fast and extensible manner. srcPat and srcQL take advantage of the speed and extensibility of XPath to provide searching technology that is friendly to developers working on source code rather than XML documents. We developed several extension functions within XPath specifically tailored for srcML. While there are other tools that provide additional support for XML querying and manipulation, we chose XPath because of its speed and wide range of support. We now describe the details of srcPat, which is then followed by a description of srcQL.

III. SRCPAT: A PATTERN MATCHING LANGUAGE

srcPat is a language for source-code patterns that takes into account syntactic structure, and is the underlying support for pattern matching in srcQL. By default, a literal piece of code is a srcPat pattern. The following srcPat expression matches a simple C++ assignment statement:

```
total = subtotal + 5;
```

Of course, this will only succeed for an exact match to the code while ignoring whitespace. For more general matching, the use of syntactic variables in place of identifiers and types can be used. The srcPat expression:

```
$U = $I + $T;
```

is a pattern generalized using *syntactic variables*. In this expression, the three variables: `$U`, `$I`, and `$T` match anything that can occur at those positions within an expression using valid (C++) syntax. In the previous literal code example, the syntactic variables `$U`, `$I`, and `$T` match to `total`, `subtotal`, and `5` respectively.

A srcPat pattern can include such things as a single statement/expression, multiple statements, a function, an entire class definition, or any other syntactic context one can express as part of a source-code pattern. srcQL makes use of srcPat when it compiles these expressions by leveraging it to create patterns containing variables, perform unification on variables between multiple patterns, as well as provide additional structural relationships between multiple srcPat expressions.

srcQL only matches the specified parts of the language being queried; it is unrestrictive. No assumptions are made about omitted parts of queries such as the contents of parameter lists or blocks. For example, take this pattern of a function definition: `void write() {}`. The pattern clearly matches occurrences of the function `write()` with no parameters. Since srcQL is unrestrictive, it will additionally match `write` functions with any number of parameters, such as `void write(int)`. It handles variable matching in a similar fashion. Variables are considered to be unspecified sections of the AST, meaning they can match anything, with restrictions placed upon their values later during unification. We chose this unrestrictive method of pattern matching because it gives flexibility in formulating generic (inclusive) queries. One can always be more restrictive by providing more information in the query. This policy allows for general searches to be performed more easily.

Syntactic variables are made to match syntactic terms that are not fully specified within the pattern and unify with other variables of the same name. Consider the following pattern: `$U* $I = new $T;` and a specific match in C++:

```
int const* numbers = new int[count];
```

The syntactic variable `$U` matches everything in the type of the declaration up to the type modifier `*`. So the value of `$U` within this match is `int const`. The syntactic variable `$I` matches the name of the code variable, `numbers`. The syntactic variable `$T` matches what occurs as part of the syntax of the statement after the use of the operator `new`. Thus `$T` doesn't just match `int`, it matches `int[count]`.

A match context is the root syntactic category (i.e., srcML element) within a srcPat pattern. This is what is used to determine where a particular pattern can occur. The match context is inferred based on the syntactic categories of the pattern and a heuristic that is used to determine which syntactic context is desired. A known list of contexts is used such as *while*, *for*, *return*, etc. For example, using the pattern `foo(){}` yields a match context of *function*, i.e., the srcML element `<function>`.

IV. SRCQL: A QUERY LANGUAGE

srcQL is a syntax-aware query language that leverages srcPat, XPath, and srcML. srcQL acts as a bridge between multiple srcPat patterns by providing search context, relational operators, and inter-pattern unification. The basic syntax for srcQL is of the form:

```
FIND search-context CONTAINS pattern
```

This is read as "Find all search contexts that contain the pattern". The *search context* is the syntactic category to be searched upon as well as what is returned as a result of the query. The search context is any valid syntactic category within the language grammar as defined in srcML (i.e., the srcML tag set). The *pattern* is in the form of a srcPath or XPath expression and describes the syntactic structure being searched for. In addition, srcQL supports the operators `WITHIN`, `FOLLOWED BY`, `WHERE`, `GROUP BY`, and `ORDER BY` to specify partial orderings, constraints on the search space, and ordering of results.

The context of a srcQL query is the syntactic category that will be matched using XPath or srcPat. The search context is where to look for matches to patterns supplied using operators as well as what is being searched for. For example, if a context is specified by the XPath `src:class`, then the only elements searched will be the children of `src:class` elements. The elements returned will be those `src:class`es that match constraints imposed by other srcQL operators.

FIND is the primary operator of a srcQL query with XPath used to specify the context. For example, to find all function/method definitions within a C/C++ system the following query is used:

```
FIND src:function
```

FIND can also accept srcPat and use the syntactic context derived from a srcPat pattern. For example, if one wishes to locate all the function definitions using srcPat instead, the following can be used:

```
FIND srcPat $R $F(){}
```

srcQL provides operators for structural relations within source code that are otherwise very difficult to describe using srcPat or XPath alone. The operator keywords are used to separate multiple srcPat expressions from one another. srcQL provides an unordered relation, a partial ordering relationship between children, a child to parent relationship, and basic predicates with regular expressions. The six srcQL operators are described below along with an example of how it is used and an explanation to clarify its behavior.

CONTAINS is an operator used to locate expressions within the search context specified by FIND. CONTAINS provides an unordered relationship between child expressions within the search context. The following query finds all functions that contain a specific usage of the C++ new operator.

```
FIND src:function CONTAINS $var = new $T
```

The result of this query is a set of functions. We can further refine the search by looking for functions that contain both a new and delete. srcQL makes this straightforward by allowing for multiple CONTAINS, as shown below:

```
FIND src:function CONTAINS $var = new $T
                  CONTAINS delete $var
```

The query finds all functions (FIND src:function) which contain an assignment to a variable with a call to the new operator (CONTAINS $var = new $T) and also has the variable matched by $var deleted later in the search context that, in this case, is what is matched by src:function (CONTAINS delete $var). In this example, CONTAINS only matches functions that contain one or more new and delete with matching $var in their expressions. Unification is used to match the $var's.

FOLLOWED BY is used to specify ordering of statements in a query. An expression matched with FOLLOWED BY is limited to occur within the sub-tree matched by the search context within document order but after the preceding CONTAINS or FOLLOWED BY. For example, to find a function which contains an *fopen*() followed by an *fclose*() on the same variable is as follows:

```
FIND src:function CONTAINS $X = fopen()
                  FOLLOWED BY fclose($X)
```

The partial ordering can be chained multiple times to match sequences of more than two statements. This is shown in section 0.

WITHIN is an operator that provides a relationship between a FIND, CONTAINS, or FOLLOWED BY, and the context in which it occurs. The pattern can reach outside of the search context specified by FIND and doesn't change the search context's scoping to operators like FOLLOWED BY. Similar to FIND, WITHIN implicitly accepts XPath and explicitly accepts srcPat expressions using the keyword srcPat. The following query will locate all function definitions within classes within the entire system:

```
FIND src:function WITHIN src:class
```

The operator WITHIN can also be applied to more than one operator of a query to specify exactly where to locate specific parts of both partially ordered and unordered expressions. This is demonstrated in section 0.

WHERE is an operator that provides a way to augment the unification process with predicates and functions that operate on the variables of srcPat. The provided predicates include regular expressions and both equality and inequality comparisons, all of which can be applied to variables during unification. This implies that if the user wants to locate a class with a specific naming convention they can use WHERE to do just that by leveraging the match:

```
FIND srcPat class $T { };
     WHERE match("^foo[0-9]", $T)
```

This will search for all classes that have a name beginning with the word "foo" followed by a number. The match srcQL function provides regular expression matching to the value of the syntactic variable.

By default, the results of a query are given in document order. ORDER BY is an operator for sorting the results of the query. It accepts a syntactic variable whose purpose is to provide a lexicographical ordering of the search results.

GROUP BY provides a simple way to gather results from srcQL, as opposed to the default document order. GROUP BY allows for a syntactic variable used in a previous operator, and also allows organizing by syntactic category of the results using a special keyword as shown below.

```
FIND srcPat class $T { };
     GROUP BY syntacticCategory
```

The results are sorted by the number of occurrences of each category. The categories for this example may include in another class, as part of a typedef, or global scope. This is a very practical and powerful feature as it provides all the syntactic situations that a given pattern occurs in. It is especially useful for helping to construct transformations or uncovering unintended usages of a construct or API call.

```
void foo3(bar* x) {
  bar* x = new bar();
  bar* a = new bar2();
  barSuperClass* i = new bar3();
  if(i)
    i->open();
  delete i;
}

class myClass {
  void foo5(bar x, bar x2) {
    delete i;
    x.open();
    if(x) {
      x.push(new bar5[4]);
    }else{
      x.pop();
}}};
```

Figure 2. Source code for example queries

V. IMPLEMENTATION

srcQL operates on srcML; an XML AST representation of the source code. It is implemented in C++ and leverages LLVM for query execution. Each query is modeled as a sequence of traversals. Each traversal consists of a set of automata that comprise a pushdown automaton. srcQL is implemented in this fashion because of how queries are evaluated, that is each child element matched is treated as a predicate to the previous match. This approach scales well compared to an earlier version we constructed in C#.

Additionally, we constructed a tool called XPathVM which also uses LLVM and operates on libxml2's XML tree representation. Basically, this is a compiler for XPath. We found that compared to the interpreted version of XPath implemented by libxml2 (the only other common option available), XPathVM is faster and uses significantly less memory. While libxml2's XPath interpreter is quite scalable for short to medium sized XPath expressions (i.e., is efficient when applied to large srcML documents) it runs into problems on large expressions, such as those generated by srcQL. Hence, the motivation for our XPath compiler.

srcQL queries are typically quite fast. A very complex query can be executed on all of *gcc* in under a minute, while trivial queries without unification, such as locating all functions that call `malloc`, take about three seconds. For example, searching for all functions which use `malloc` followed by `free` takes about nine seconds to complete. These timing are for a laptop computer.

The srcQL tool is made available through both a command line tool as well as through a library interface. The library interface provides a more complete and powerful interface that allows for further interaction with the source code. While the command line interface is a simple tool that allows the users to search on srcML archives or folders of archives. srcQL will be available for download from srcML.org.

VI. USING SRCQL

We use the source code in Figure 2 to demonstrate several srcQL queries. The first example of a query is to locate each expression that uses a specific operator. For example, to retrieve all expressions that contain the operator `new`, the srcQL is:

```
FIND new $T
```

When applied to the source code in Figure 2, the results match all uses of the `new` operator, including: `new bar()`, `new bar2()`, `new bar3()`, and `new bar5[4]`. The syntactic match of `$T` is the operand of the `new` expression. The example query "*Locate all if-statements within a program*" can be written as follows:

```
FIND srcPat if() { }
```

The empty condition and body matches all if-statements. This query also locates if-statements without a block because the block in srcPat matches any nested statement. The query will match the following two if-statements when applied to the source code in Figure 2:

```
if(i)
  i->open();
if(x) {
  x.push(new bar5[4]);
}else{
  x.pop();
}
```

The query "*Find all of the expression statements which use a new operator*" can be achieved by specifying the search context for the query as below:

```
FIND src:expr_stmt CONTAINS new $T
```

For the code in Figure 2, the result is:

```
x.push(new bar5[4]);
```

Take the query "*Find all pointer declarations initialized with a `new` operator with the same type in the pointer declaration as in the new expression*". srcPat provides unification that can easily handle queries such as:

```
$T* $Var = new $T
```

Producing the result:

```
bar* x = new bar()
```

Another query that demonstrates `WITHIN` is "*Get all `void` member definitions that occur within classes and have at least two parameters.*" Note the two syntactic variables, `$P1` and `$P2` which match the parameters.

```
FIND srcPat void $Func($P1, $P2) { }
     WITHIN srcPat class $ClassName { };
```

Producing the result:

```
void foo5(bar x, bar x2) {
  delete i;
  x.open();
  if(x) {
    x.push(new bar5[4]);
  }else{
    x.pop();
  }}
```

This next query demonstrates how to locate all of the declarations of pointers where the name of the type matches a regular expression. In this case the query is "*Find all declarations where the type has the name that ends in SuperClass.*"

```
FIND srcPat $T* $Var
     WHERE match("^.*SuperClass", $T)
```

Producing the result:

```
barSuperClass* i = new bar3();
```

Complicated queries can be formulated that take advantage of unification to constrain the search. For example, "*Find all functions containing a variable that is initialized using new and is opened within an if-statement that checks it and then followed by that variable being deleted".*

```
FIND src:function CONTAINS $X* $I = new $T
                  FOLLOWED BY $I->open()
                  WITHIN srcPat if($I) {}
                  FOLLOWED BY delete $I
```

Producing the result:

```
void foo3(bar* x) {
  barSuperClass* i = new bar3();
  if(i)
    i->open();
  delete i; }
```

VII. Related Work

TXL [5], RASCAL [6], DMS [7], ELAN [8], Stratego [9], and Coccinelle [10] are all program transformation languages. None are intended to explicitly support matching of source-code patterns, but rather they support the activity for conducting program transformation. Thus, their syntax and operators are designed with transformation in mind, which makes it difficult to support the unrestrictive nature of srcQL's patterns as well as operators that constrain syntax orderings. With the exceptions of Coccinelle and TXL, they require full compilation. SOUL [11], and Rscript [12] provide functional languages for matching and transformation. Browse-By-Query (BBQ) (http://browsebyquery.sourceforge.net/) allows the user to specify queries using a semi-structured natural language description of the programs structure. SOUL, RScript, SemmleCode [13], .QL [14], Java Tools Language (JTL) [15], JrelCal [16], and CrocoPat [17] all require full compilation in order to execute a query. JTL, however, does not require explicit knowledge of the internal AST. In order to do more complex queries additional functional programming is used. JTransformer [18] performs querying in order to provide aspect weaving into a Java program, and as such requires the AST.

The tools mentioned above are not all explicitly for source-code querying but their users do have to accomplish some sort of matching. srcQL removes the need for developers to learn complicated, low-level details of programming-language grammar or expression-matching techniques.

VIII. Conclusions & Future Work

The paper presents the design and implementation of a new, syntax-aware query language for source code. srcQL's application to software-engineering problems is broad. There are many examples of adaptive maintenance in which srcQL is of benefit to developers. We believe that one such task is program transformation. As stated earlier, many languages already support transformation, but do not separate the matching and transformation tasks. Part of our future work is to combine srcQL with another srcML-based language that performs transformation. The increased ease of matching source-code elements will help lower the bar for applying program transformation to adaptive-maintenance tasks. The advantage of srcQL here is the ability to find all of the syntactic variations of a given expression so that the developer can be aware of all cases. Thus, they will be able to address all situations that need to be transformed. The only alternative option currently is to search through code manually. srcQL can also be used to pinpoint all situations that a transformation needs to be applied in a programmatic manner, allowing transformations to be applied directly to the result of a srcQL query.

One limitation of the language is the current inability to nest or combine multiple queries into a single query and perform set operations upon results of multiple queries. This would allow for simplifying the formulation of some complex queries. We intend to add set operations and nesting of queries in the future. Currently, we are working on a stable implementation that addresses runtime and scalability issues, along with adding some additional functionality. This will make the language very usable within an IDE. The tool will be open sourced as part of the srcML infrastructure (www.srcML.org). This work is supported in part by a grant from the US National Science Foundation CNS 13-05292/05217.

References

[1] M. L. Collard, M. J. Decker, and J. I. Maletic, "Lightweight Transformation and Fact Extraction with the srcML Toolkit," in *the Proceedings of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2011, pp. 173-184.

[2] J. I. Maletic, M. L. Collard, and A. Marcus, "Source Code Files as Structured Documents," in *Proceedings of 10th International Workshop on Program Comprehension (ICPC)*, 2002, pp. 289-292.

[3] M. L. Collard, Decker, M. Maletic, J.I., "srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code," in *the Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM)*, Eindhoven, The Netherlands, 2013, pp. 22-28.

[4] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A Brief Survey of Program Slicing," *ACM SIGSOFT Software Engineering Notes,* vol. 30, pp. 1-36, 2005.

[5] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider, "Software engineering by source transformation - experience with TXL," in *the Proceedings of First IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, 2001, pp. 168-178.

[6] P. Klint, T. van der Storm, and J. Vinju, "Rascal: A domain specific language for source code analysis and manipulation," in *the Proceedings of the Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2009, pp. 168-177.

[7] I. D. Baxter, C. Pidgeon, and M. Mehlich, "DMS®: Program transformations for practical scalable software evolution," in *the Proceedings of the 26th IEEE International Conference on Software Engineering (ICSE)*, Edinburgh, Scotland, UK, 2004, pp. 625-634.

[8] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek, "ELAN: A logical framework based on computational systems," *Electronic Notes in Theoretical Computer Science,* vol. 4, pp. 35-50, 1996.

[9] E. Visser, "Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5," in *Rewriting techniques and applications*, 2001, pp. 357-361.

[10] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in Linux device drivers," *ACM SIGOPS Operating Systems Review,* vol. 42, pp. 247-260, 2008.

[11] C. D. Roover, C. Noguera, A. Kellens, and V. Jonckers, "The SOUL tool suite for querying programs in symbiosis with Eclipse," in *the Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, Kongens Lyngby, 2011, pp. 71-80.

[12] P. Klint, "How Understanding and Restructuring Differ from Compiling " A Rewriting Perspective," in *the Proceedings of the 11th IEEE International Workshop on Program Comprehension (ICPC)*, Portland, Oregon, USA, 2003, p. 2.

[13] M. Verbaere, E. Hajiyev, and O. D. Moor, "Improve software quality with SemmleCode: an eclipse plugin for semantic code search," in *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented programming systems and applications companion Companion SIGPLAN*, Montreal, Canada, 2007, pp. 880-881.

[14] O. Moor, D. Sereni, M. Verbaere, E. Hajiyev, P. Avgustinov, Torbj\, *et al.*, ".QL: Object-Oriented Queries Made Easy," in *Generative and Transformational Techniques in Software Engineering II*, Lammel, R. Joost, V. Eds., ed: Springer-Verlag, 2008, pp. 78-133.

[15] T. Cohen, J. Gil, and I. Maman, "JTL: the Java tools language," in *Object-oriented programming systems, languages, and applications (SIGPLAN)*, Portland, Oregon, USA, 2006, pp. 89-108.

[16] P. Rademaker, "Binary relational querying for structural source code analysis," M.S. Thesis, Netherlands, University Utrecht, 2008.

[17] D. Beyer and C. Lewerentz, "CrocoPat: Efficient pattern analysis in object-oriented programs," in *International Workshop on Program Comprehension (ICPC)*, Portland, Oregon, USA, 2003, pp. 294-295.

[18] G. Kniesel and U. Bardey, "An analysis of the correctness and completeness of aspect weaving," in *the Proceedings of the 13th IEEE International Working Conference on Reverse Engineering (WCRE)*, Benevento, Italy, 2006, pp. 324-333.