SRCQL: A SYNTAX-AWARE QUERY LANGUAGE
FOR EXPLORING SOURCE CODE

A thesis submitted
to Kent State University in partial
fulfillment of the requirements for the
degree of Masters of Science

by
Brian Bartman
Summer 2013
Thesis written by

Brian Bartman

B.S., Kent State University, USA 2011

Approved by

Jonathan I. Maletic, Advisor

Javed Khan, Chair, Department of Computer Science

Raymond Craig, Dean, College of Arts and Sciences
# Table of Contents

LIST OF FIGURES ............................................................................................................................ V  
LIST OF TABLES ............................................................................................................................... VI  
DEDICATION ......................................................................................................................................... VII  
ACKNOWLEDGEMENTS ....................................................................................................................... VIII  

CHAPTER 1 INTRODUCTION ............................................................................................................. 1  
  1.1 Contributions ............................................................................................................................... 3  
  1.2 Organization of Thesis .................................................................................................................. 3  

CHAPTER 2 INFRASTRUCTURES USED IN RESEARCH ................................................................. 4  
  2.1 srcML ........................................................................................................................................ 4  
  2.2 XPath ......................................................................................................................................... 6  

CHAPTER 3 SRCPAT: A PATTERN MATCHING LANGUAGE FOR SOURCE  
  CODE ................................................................................................................................................. 7  
  3.1 Pattern Matching .......................................................................................................................... 8  
  3.2 Variable Matching ....................................................................................................................... 9  
  3.3 Match Context Inference ............................................................................................................ 11  
  3.4 Unification .................................................................................................................................. 13  
  3.5 Limitations of srcPat Unification ............................................................................................... 15  

CHAPTER 4 SRCQL: A SYNTACTIC QUERY LANGUAGE ............................................................... 17  
  4.1 SELECT Context ........................................................................................................................... 17  
  4.2 srcQL Operators .......................................................................................................................... 19  
    4.2.1 CONTAINS ............................................................................................................................ 19
LIST OF FIGURES

Figure 3.1 srcPat unification string comparison function.............................................. 14
Figure 3.2 srcPat unification algorithm ................................................................. 14
Figure 4.1 Operator Relationship Tree ........................................................................ 29
Figure 4.2 EBNF grammar for srcQL ............................................................................ 35
Figure 5.1 Source code for example queries ................................................................... 41
LIST OF TABLES

Table 4.1 Unification table built from the srcPat pattern $X = fopen(). .................. 32

Table 4.2 Unification table built from srcPat pattern fclose($X). .......................... 32

Table 4.3 Unification table built from srcPat pattern fputs($Str, $X). .................. 32

Table 4.4 Intermediate unification of Table 4.1 and Table 4.2................................. 33

Table 4.5 Intermediate unification showing the union of Table 4.3 and Table 4.3 ....... 34
DEDICATION

I'd like to dedicate this paper to my dog Gus and my mother for supporting me while I complete my education.
ACKNOWLEDGEMENTS

I’d like to thank the committee and my advisor for their help and effort for helping me complete my thesis.

Brian Bartman

November, 1 2013. Kent, Ohio
CHAPTER 1

Introduction

Software developers spend a great deal of time exploring and searching source code. Exploration is done to help comprehend the purpose and behavior of a software system. For example, when a feature needs to be modified in a system, the developer(s) must explore the source code and locate where the feature is actually implemented. Tracking down the specific location of a bug also requires exploration. Given the exponential growth in the size of software systems over the past 20 years, exploration has become extremely difficult. As software systems grow into millions of lines of code, existing search and exploration tools, such as grep, fail to support developer needs. The main drawback with regular expression or text based exploration 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 should be inherently aware of the underlying syntax of the programming language. We propose, implement, and demonstrate a syntactic aware query language that is built on top of a syntactic aware pattern matching language. The query language is called srcQL (source Query Language) and the pattern matching language is called srcPat (source Pattern matching).
The objective of this research is to allow developers to easily construct queries that take into account the syntactic features of the programming language. For example, a developer may want to find all the functions in a system that have a call to both open and close with the body of the function. Furthermore, the ordering may be of interest; open before close. This could 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 a regular expression.

Currently, there are no languages specifically designed for syntactic aware source code querying. However, there are languages that support program transformation to various degrees that do some limited types of syntactic pattern matching. None of these approaches have been adopted due to the complexity of the language or the requirements of understanding fine detail about the grammar for the language. In short, they are difficult and awkward to use. Furthermore, they typically work on the abstract syntax tree (as generated by the compiler) and as such do not map one-to-one with the actual source code. This tree based matching is also extremely inefficient and as such cannot be practically applied in real time within the context of an IDE (Integrated Development Environment).

To overcome these roadblocks we built the query language on top of the srcML infrastructure [Collard, Decker et al.] and leverage XML technologies (i.e., XPath). srcML is has extremely scalable parsing technology that marks up source code with abstract syntactic information in XML tags. This addresses, to a large part the scalability and efficiency problems. Moreover, we modeled srcQL after the database query
language SQL. This makes developing and understanding the meaning of srcQL queries fairly natural for most developers (at least those familiar with SQL).

1.1 Contributions

The design and implementation of a novel syntactic aware query language for C++ is presented and demonstrated. The language srcQL is built on top of a syntactic aware pattern matching language, srcPat. The language is compiled into XPath and then applied to srcML (XML) representations of source code programs. This is the only source code query language that is efficient, scalable, and relatively natural to use.

1.2 Organization of Thesis

This thesis is organized as follows. CHAPTER 2 describes the infrastructure that is used to support the construction of srcPat and srcQL. CHAPTER 3 presents srcPat pattern matching language and CHAPTER 4 presents the srcQL query language. CHAPTER 5 gives an overview of the implementation of srcQL and srcPat, along with a set of examples to demonstrate the tools. Related work is presented in CHAPTER 6, and conclusions are presented in CHAPTER 7 along with future work.
CHAPTER 2

Infrastructures Used in Research

This chapter talks about the different infrastructure used to support the creating of the srcPat and srcQL languages. We describe the different tools that were chosen in order to implement both languages. We discuss the different needs of each of the tools used within our research as well as the requirements that they fulfill.

2.1 srcML

srcML\(^1\) (SouRce-Code Markup Language) [Collard, Decker et al. 2011] is an XML format used to augment source code with syntactic information from the AST to add explicit structure to program source code. All original text of the source code, including comments, preprocessing information, and formatting, is preserved and identified for use by program-comprehension tools and development environments. The focus is to construct a document representation in XML instead of a traditional source-code document or data representation. This representation supports a programmer-centric, rather than compiler-centric, view of the source code.

The srcML format is supported by a toolkit, src2srcml and srcml2src, which include conversion between source code and the format. The format and translation tools for srcML have in the past been used for searching source code via any tool that supports XML manipulation or querying.

\(^1\) http://www.srcml.org/
Complete View of the code: srcML takes an unprocessed view of the source code. This means that the source code can have both structure and textual information. That is how programmers generally view their code.

Robustness: srcML is able to represent code that is not compilable. The src2srcml tool is able to convert incomplete code even when there are translation problems. This only affects the quality of the markup and does not lead to a loss of any of the original text.

Efficiency: The tool is very efficient with a translation speed of 25 KLOC per second and can handle almost 3,000 files per minute, e.g., the entire Linux kernel can be converted to the srcML format in less than seven minutes. Going from srcML back to source code is handled by the tool srcml2src, which is even faster with speeds over 250 KLOC per second.

Independence: The srcML translator is not based on the parser of a particular compiler. It contains its own parser that only needs to understand enough about the code to insert the proper markup. Because it stops at this markup, it can allow syntax that a particular compiler cannot.

XML: The srcML format is an XML representation, and is designed to take advantage of current and future XML tools for transformation, validation, etc. To date the format has been successfully used with XSLT, DOM, and SAX approaches.

srcML is used by srcPat to derive the syntactic elements of a search pattern. The pattern is used to generate an XPath representation of that pattern. Each pattern is then
used to compose a larger XPath with all relationships given by srcQL, which is used to query source code. srcQL bridges the gap between the source code and the srcML representation by providing more source code centric means of describing a query on source code.

2.2 XPath

XPath is a standardized way of querying XML documents, which is both fast and extensible. srcPat and srcQL take advantage of the speed and extensibility of XPath in order to provide a speedy search and to allow XPath to better understand source code relationships within the srcML representation using extension functions within XPath.

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 considered several other tools to implement srcPat and srcQL such as LINQ, relaxNG and XQuery, but they didn’t provide the necessary speed and portability of XPath queries.
We now describe srcPat, the underlying support for pattern matching in srcQL. srcPat is a language for pattern matching of source code that takes into account syntactic structure. 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 in C++, generalized using syntactic variables. In this expression, the three variables: $U, $I, and $T match anything (i.e., valid syntactic category) 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 would match: total, subtotal and 5 respectively.

A srcPat pattern can range in size including a single statement/expression, multiple statements, a function, an entire class definition, or any other syntactic context.
one can express as part of their source code pattern matching. srcQL makes use of srcPat when it compiles these expressions by leveraging it in order to create patterns containing variables, performing unification on variables between multiple patterns as well as providing additional structural relationships between multiple srcPat expressions.

3.1 Pattern Matching

The matching process of srcPat is unrestrictive when it comes to parts of the grammar that are not present within a search expression, but restrictive when it comes to those which are present. For example, if we consider the srcPat example, \( A + B \) and we generalize that to be,

\[
\$V1 + \$V2
\]

where the two variables can be any two syntactic elements, that occur within the language, separated by an operator +, such as two variable names or even entire expressions.

srcPat doesn’t assume anything about omitted parts of queries such as the contents of parameter lists, blocks and alike. One example of this is the following pattern of a function definition,

\[
\text{void write() \{ \}}
\]

The pattern matches occurrences of the function \text{write() \} with no parameters. The pattern is also unrestrictive in that it will also match write functions with any number of parameters

The pattern can include parameters, as in:
void write(FILE*) { }

This pattern matches all definitions of the write functions that return void and the type of the first parameter is FILE*. It also matches all write functions with more than one parameter, where the first parameter has the type FILE*. Because of the unrestricted policy of srcPat there is no need to supply a variable name because the only thing whose existence is searched for is FILE* srcPat will not check for the non-existence of a variable name.

We chose this unrestricted method of pattern matching because gives quite a bit of flexibility in formulating generic (inclusive) queries. Also, one can always be more restrictive by providing more information in the query. This policy allows for general searches to be performed more easily.

3.2 Variable Matching

Variables are made to match syntactic items that are not fully specified within the pattern. A pattern of only a single variable, e.g., $V$, will match every non-keyword syntactic element within the program. This is because a single syntactic variable can represent anything other than a keyword within the program. Consider the following expression:

$U* \ I = \text{new} \ T$

and a specific match in C++:

int const* numbers = new int[count];
The 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 variable \$I matches the name of the variable declaration, which makes \$I match numbers\. The variable \$T matches the syntax that occurs as part of the syntax after the use of the operator new\. Thus \$T doesn’t just match int, it matches int[count]\.

Variable matching of parameters of functions is done slightly differently\. If a variable occurs within a function declaration or definition the match is based upon how many variables occur within a single argument\. For example, the following srcPat expression:

```c
void $FuncName($ParameterOne) { }
```

matches a function definition with at least one or more parameters\. Because there is only one variable given for the parameter, the entire expression that occurs within the first parameter is matched and not just the type of the first parameter\. In order to match the syntactic variable type and variable name of a parameter expression two variables need to be used\. In order to match both the parameter type and name the following pattern can be used:

```c
void $FuncName($ParameterOne $VarName) { }
```

Variables can also be used to match the contents of entire block of source code, although a variable is not required to match a non-empty block\. In order to match the content of an entire block within C++, one would simply place a variable within the block without a semi-colon as shown in the following srcPat expression:
void $FuncName($ParameterOne) {
    $BlockContent
}

This will match $BlockContent to all the statements of any void function with at least one parameter.

3.3 Match Context Inference

A match context is the root XML element within a srcPat pattern. This is what is used to determine where a particular pattern can occur. In srcPat, the match context is inferred from the statement or expression based on the syntactic context in which they occur and a heuristic that is used to determine which syntactic context is desired. To accomplish this, a list of known contexts is kept and referenced.

These known contexts are the syntactic contexts from the language grammar of constructs such as `while, for, return`, etc. In srcML, these correspond to the names of corresponding XML elements. The match contexts are the nodes of the abstract syntax tree. To find the context of a srcPat expression we simply traverse from the root of the srcML document down to one of these context nodes. For example, take the following code statement:

```
int foo(){}
```
The function `foo()` is marked up with the context node `<function>` in srcML. The element `<function>` is in the list of possible match contexts, so the match context of the srcPat pattern is selected to be `<function>`.

There other potential context nodes within the list. Only two of them require special treatment, namely expression and declaration statements. Expression and declaration statements are handled in the same way. If there is a semi-colon, it is processed as a statement and if there isn’t a semi-colon than the expression or declaration is processed as an expression and declaration respectively, instead of a statement. The reason for the semi-colon aware behavior is that it allows specification of parts of an expression or declaration statement. For example consider the following pattern,

```plaintext
foo();
```

This pattern will locate all of the expressions statements which call the function `foo`. For example, the above srcPat will match the `foo` within the function `bar` in the following example,

```plaintext
int bar() {
    foo();
}
```

However the following call to `foo` will not be matched.

```plaintext
int bar() {
    if( foo() ) { } 
}
```
This is because the semi-colon within the search pattern limits the syntactic context of the pattern expression statements instead of just expressions. If instead we omitted the semi-colon from the pattern, both calls to \texttt{foo} will match.

### 3.4 Unification

For srcPat variables of the same name within a srcPat expression, e.g., \$T = $T + 5, intra-pattern unification is performed. The algorithm for unification is given in Figure 3.2. Unification within srcPat is done heuristically using a greedy approach where the most general unifier is taken. Matches to variables within a pattern are first normalized into a string where the value of each XML element is taken and the whitespace between each element and child element is normalized. This normalization provides a means of comparing parts of the XML document that would otherwise be represented as different parts of syntax, such as the type of a variable declaration, the type used in a constructor call to operator new, or a class name. After each of the variable values is normalized, the values are compared starting with the first pair, using a “starts with” string comparison. That is, if the first variable’s value starts with the other’s value or the second variable’s value starts with the first, than the variables are said to match. In the event of a matching variable, the shorter string is kept for further comparison and the process is continued using the string resulting from the previous comparison and the next value of a variable.

```java
1. normalizeSynVar(SynVarMatch)
```
2. result = ""  
3. for element in SynVarMatch  
4.     for text in element.getAllTextNodes()  
5.         if text is not whitespace  
6.             result.append(text + " ")  
7. return result

Figure 3.1 srcPat unification string comparison function.

1. srcPatUnification(SynVars)  
2.     currentMatch = normalizeSynVar(SynVars[0])  
3.     for i = 1; i < SynVars.length; ++i  
4.         normalizedVar = normalizeSynVar(SynVar[i])  
5.         if( currentMatch.startsWith(normalizedVar) ||  
6.             normalizedVar.startsWith(currentMatch))  
7.             if normalizedVar.length< currentMatch.length  
8.                 currentMatch = normalizedVar  
9.         else  
10.             return false  
11.     return true

Figure 3.2 srcPat unification algorithm.

Given the above algorithms consider the following srcPat pattern and match of that pattern. The following pattern searches for a variable where both of the $T$ match.

$$T* I = new T$$

An example match would look like:

foo* var = new foo() 

The unification gives two possible values for $T$ (quotes added for readability):
"foo "

"foo ( )"

The normalized values of $T$ shown above are then compared using the unification algorithms and decided to be a match because "Foo ( ) " starts with "Foo ". The reason for the decision to use this algorithm instead of directly comparing the syntax trees that exist within the XML is that, in simple cases, such as the one listed above, are syntactically different, but textually similar and logically the same. The reason for using “starts with” as a loose comparison instead of using a more restrictive comparison such as a complete string comparison is that, the string equals comparison tends to be more restrictive than necessary and it normally exclude some of the logical matches such as in the example given above, without excluding any of the exact matches.

3.5 Limitations of srcPat Unification

There are several limitations on the abilities of unification using our simple naïve comparison algorithm. One of those limitations is that during the comparison of variables, things such as logical not or parenthesis can exclude some of the logic matches, because they are textually considered dissimilar. For example, if the user searches for an expression within two places using the following pattern:

```c
FILE* $X = fopen();
if($X) {
}
```

and matched
FILE* log = fopen();

if(!log) {
    }

The match would be rejected during unification because of the "!X " within the if statement.
CHAPTER 4

srcQL: A Syntactic 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:

```
SELECT search-context CONTAINS pattern
```

This is read as “Select 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. The *pattern* describes the syntactic structure being searched for. The pattern can be any srcPat or XPath expression. In addition, srcQL supports operators including *WITHIN, FOLLOWED BY, WHERE, GROUP BY, and ORDER BY*. These provide a way to specify such things as partial orderings and constraints on the search space. The context as well as each operator is now described in more detail.

4.1 SELECT Context

The context of a srcQL query is the syntactic category that will be searched for and matched using XPath or srcPat. The search context refers to the node (in the abstract
syntax tree) being selected by an XPath or srcPat expression. The root node can either come from the user as part of an XPath or be inferred from a srcPat expression. 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:classes that match criteria provided by other operators.

SELECT is the beginning operator of a srcQL query and it expects XPath to specify the context. For example, to select all function/method definitions within a C/C++ system one would use the following expression,

SELECT src:function

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

SELECT srcPat $R $F(){}

A similar expression can be used which allows SELECT to simply be given a * and then SELECT uses the syntactic context from CONTAINS. For example the following produces the same result as the previous query:

SELECT * CONTAINS $R $F(){}
4.2 srcQL Operators

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 various srcPat expressions from one another.

The srcQL language consists of six operators. A description of each is given below along with an example of how it is used and an explanation to clarify its behavior. Some of the operators are drawn directly from SQL while others came about because we are querying a tree and not a table. The srcQL language provides an unordered relation, a partial ordering relationship between children, child to parent relationship, and basic predicates that can use regular expressions. We now describe each of these operators in detail.

4.2.1 CONTAINS

CONTAINS is an operator used to locate multiple different expressions within the search context specified by SELECT. CONTAINS provides an unordered relationship between child expressions within the search context.

We will now look at a few examples; the following query finds all functions that contain a certain usage of the new operator.

```
SELECT 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 simple by allowing for multiple CONTAINS, as shown below:

```sql
SELECT src:function
CONTAINS $var = new $T
CONTAINS delete $var
```

The query finds all functions (SELECT 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 was matched by src:function (CONTAINS delete $var).

CONTAINS, in the above example, behaves such that it only matches functions that contain one or more ‘new’ and ‘delete’ keywords with matching $var in their expressions.

The following query is an example of using XPath instead of the default srcPat. It finds the use of operator new within a declaration statement:

```sql
SELECT src:decl_stmt
CONTAINS XPath op:operator[.="new"]
```

Because CONTAINS supports an unordered relationship among expressions within the SELECT context, multiple CONTAINS can be used while describing one syntactic context.
If two 

CONTAINS

operators are used that both search for the same expression, the parts of the AST they match are not restricted to being different, meaning that they can both match the same nodes within the AST. For example

```
SELECT src:function
CONTAINS write()
CONTAINS write()
```

Will have the same result as,

```
SELECT src:function
CONTAINS write()
```

This means that 

CONTAINS

cannot be used to search for cardinality of an expression within a context. Providing two CONTAINS to the same search context with the same pattern only matters when you add additional descriptive operators to them such as 

FOLLOWED BY

or 

WITHIN

which are described later. One implication of this is that 

CONTAINS

cannot be used to provide a count operation to a particular pattern within a search context.

### 4.2.2 FOLLOWED BY

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:

```sql
SELECT 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. For example, to locate a function which first called `fopen()`, followed by `fprintf()` then followed by `fclose()` is done with the following query:

```sql
SELECT src:function
CONTAINS $X = fopen()
FOLLOWED BY fprintf($X)
FOLLOWED BY fclose($X)
```

4.2.3 WITHIN

WITHIN is an operator that provides a relationship between a SELECT, CONTAINS or FOLLOWED BY and the context in which it occurs. The ancestor can reach outside of the search context specified by SELECT and doesn’t change the search context’s scoping to operators like FOLLOWED BY.

Like SELECT, WITHIN implicitly accepts XPath and explicitly accepts srcPat expressions using the keyword srcPat. So if one wanted to locate all function definitions within classes within the entire system it can be easily achieved:
SELECT 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. For example, if one were to search for functions that uses a basic C style I/O pattern of fopen, check to make sure stream is open, fprintf and fclose, as seen in the following example,

```c
void outputData() {
    FILE* outputFile = fopen('w');
    if(outputFile) {
        printf(outputFile, "data");
        fclose(outputFile);
    }else{
        return;
    }
}
```

the srcQL query that usage pattern can be located:

```
SELECT src:function CONTAINS $X = fopen()
FOLLOWED BY fprintf()
WITHIN srcPat if($Z) { }
FOLLOWED BY fclose($Z)
```
WITHIN doesn’t change the location of the next FOLLOWED BY, so the call to fclose could be within the if($X) { }, after it, or within an else block. It is currently not possible to specify which block, either the if or the else but, the fprintf will occur within at least one of them. In the future using additional operators it will be possible to do so.

4.2.4 WHERE

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, 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 could use WHERE to do just that by leveraging the match:

```
SELECT 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 function provides regular expression syntax to the syntactic variable’s values to allow the programmer to refine their search further. The matching is done on a pretty printed string; this means that if there was an export macro it would be included in the name of $T and would also have to be matched, but it would always be separated by a single space.
4.2.5 ORDER BY

ORDER BY is an ordering clause that allows for alphanumeric type sorting of the results of the query. It accepts a syntactic variable for which provides a total ordering of all search results returned to the user or a user provided function to provide a total ordering of groups. A query to get all classes in the archive and order them by name is used as an example.

```
SELECT srcPat class $T { }; ORDER BY $T
```

The ORDER BY can occur in one of two places. The first is to provide a total ordering and the second is after a GROUP BY in order to provide a group ordering in the event that ORDER BY is used to provide an ordering for groups with a Count() function or a user provided extension function. For example, to order the groups of a GROUP BY the largest to smallest:

```
SELECT srcPat class $T { }; 
GROUP BY $T 
ORDER BY Count()
```

If a total ordering is not given then the default is to use document ordering. In the event that a group ordering isn’t given then item groups are organized by which occurred first within the document.
4.2.6 GROUP BY

GROUP BY provides a simple way to gather and sort results from srcQL. By default, srcQL outputs results in document order, i.e., the order they occur in the source code. GROUP BY accepts a syntactic variable used in a previous SELECT. For example, to sort by function name, the query is:

```
SELECT srcPat $R $F() { }; GROUP BY $F
```

All of the functions are grouped by function name. To change the sort order of the groups, an ORDER BY can be used:

```
SELECT srcPat $R $F() { }; 
GROUP BY $F ORDER BY Count()
```

The groups within the above example refer to all of the unique values of $F. This will sort the groups from largest to smallest before returning them.

In addition to taking a syntactic variable name, the clause also allows for a grouping function, or the keyword syntacticCategories.

When “GROUP BY syntacticCategories” (described in the Application of srcQL section) is used, no other ordering can be given with it. All syntactic categories are ordered by the Count() function. The following is an example of how to use the syntactic category ordering of results:

```
SELECT srcPat class $T { }; 
GROUP BY syntacticCategories
```
The API for srcQL allows for additional functions to be added in addition to the function `Count()`.

### 4.3 Unification

srcQL uses variables for matching to the syntactic structure and to perform pattern matching where relationships between sub-expressions needs to be specified. Unification between multiple srcPat patterns within a srcQL statement is taken care of in srcQL. The matching of the variables is accomplished via a unification process.

srcQL unification is a multistep process whereby variables of the same name are compared against one another. The comparisons are made between all srcPat expressions that contain variables. The steps for unification are as follows:

1. Gather all possible matches into a match candidate set by executing a query without performing unification and perform steps 2 through 5 on each one.
2. Variables within the same expression are compared;
3. several tables are built from the possible match trees, based upon operator relationships show in Figure 4.1 from within a match with the variables as columns and possible values for the variables as rows; then
4. The tables are joined on each variable name and compared if the variable values match using the algorithm from Figure 3.2 than the shorter of the two values is kept for further comparison with other values, this is done until only a single table remains or a table with no rows is encountered; then
5. The predicate expressions from WHERE are evaluated against each row. If at any point during unification a table has no rows in it, the match is rejected and the next candidate match is processed.

Not all of the operators within a single match participate in unification. The general rule is that if an operator contains a variable that occurs more than once throughout the srcQL query than it is used during unification, otherwise it does not add any additional computation. Because of the large increase in processing requirements, unification can be skipped entirely if there is no WHERE operators and all variables are only used once.

The relationships between operators show in Figure 4.1 depicts the different relationships of multiple operators within a query. The individual matches within a match of a srcQL query are then used to form the tables.
Because the individual operators can match one or more syntactic elements within a possible match candidate each possible value for those operators is considered as a possible match for that pattern, it alters the number of matches that are part of the unification process. This changes the relationship from simply being a 0..1 relationship within the srcQL language to being a 0..* relationship within a match candidate. For example, take the simple query

```sql
SELECT srcPat $X.foo();
WITHIN srcPat if($X) { }
```

On the following source code,

```java
if(alpha) {
    if(beta) {
        if(theta) {
            alpha.foo();
        }
    }
}
```

Figure 4.1 Operator Relationship Tree
The source code has match candidate a line 4, and three matches to **WITHIN** at lines 1, 2 and 3. This simple example shows the possibility of three one for each of the operators `srcPat` expression. Each of the lines 1, 2 and 3 are gathered and the variable value of `$X` is extracted and compared to the variable value of the `$X` that is contained upon line 4. Line 4 matched by `SELECT` forms a single table of one row and one column and the **WITHIN** forms a table to three rows and one column, then both of the tables are joined on their one column `$X`, yielding a single table of one row and one column.

Unification with a **CONTAINS** can be much more and tables because of its relationship to both **FOLLOWED BY** and **WITHIN**. When a **CONTAINS** builds a tree with all of its **FOLLOWED BY** clauses and their **WITHINs**, unification is done by individually selecting each of the possible matches within the context, and each of **WITHIN** matches are than paired with each possible match for the **CONTAINS** and **FOLLOWED BY** clauses. Each possible **FOLLOWED BY** match is located as they relate to each possible **CONTAINS** match. Then each possible **WITHIN** is paired with each individual **FOLLOWED BY** clause match. This process is repeated for each of the subsequent **FOLLOWED BY** clauses, where each **FOLLOWED BY** match is computed relative to its previous **FOLLOWED BY** instead of relative to **CONTAINS** like the first **FOLLOWED BY**. Each possible match set from **CONTAINS** down to the last **FOLLOWED BY** clause create a single row match within the table. Each row within the table is condensed by comparing it against the syntactic variable values of each other row for uniqueness before it is inserted into the table.
The following is an example of the unification process. Consider the following srcQL query and match for it:

```
SELECT src:function
CONTAINS $X = fopen()
CONTAINS fclose($X)
CONTAINS fputs($Str, $X)
WHERE match("\"match stuff\"", $Str)
```

A possible match for the srcQL query is the following C/C++ source code:

```c
void foo() {
    otherfilePtr = fopen("output.txt","w");
    filePtr = fopen("output.txt","w");
    ... 
    fputs("write stuff", filePtr);
    ... 
    fclose (filePtr);
    fclose (otherfilePtr);
}
```

The function matched by the existence of three of its clauses that contain syntactic variables. Each of CONTAINSs are used to generate tables, shown below:
Table 4.1 Unification table built from the srcPat pattern $X = \texttt{fopen}()$.

<table>
<thead>
<tr>
<th>Code</th>
<th>Syntactic variable: $X$</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>otherfilePtr = fopen (...)</code></td>
<td><code>otherfilePtr</code></td>
</tr>
<tr>
<td><code>filePtr = fopen (...)</code></td>
<td><code>filePtr</code></td>
</tr>
</tbody>
</table>

Table 4.2 Unification table built from srcPat pattern `fclose($X)`.

<table>
<thead>
<tr>
<th>Code</th>
<th>Syntactic variable: $X$</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fclose(filePtr)</code></td>
<td><code>filePtr</code></td>
</tr>
<tr>
<td><code>fclose(otherfilePtr)</code></td>
<td><code>otherfilePtr</code></td>
</tr>
</tbody>
</table>

Table 4.3 Unification table built from srcPat pattern `fputs($Str, $X)`.

<table>
<thead>
<tr>
<th>Code</th>
<th>$Str</th>
<th>$X</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fputs(&quot;write stuff&quot;, filePtr)</code></td>
<td>&quot;write stuff&quot;</td>
<td>filePtr</td>
</tr>
</tbody>
</table>
The next step within unification is to perform joins on syntactic variables of the same name, so in this case the only variable which is shared between any two clauses is $X$ and because $Str$ is needed by the where clause, it earns a spot in the table.

### Table 4.4 Intermediate unification of Table 4.1 and Table 4.2

<table>
<thead>
<tr>
<th>Intermediate Table 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntactic variable:</strong> $X</td>
</tr>
<tr>
<td>filePtr</td>
</tr>
<tr>
<td>otherfilePtr</td>
</tr>
</tbody>
</table>

Then when the intermediate Table 4.4 is joined with Table 4.3 it yields the final table before the **WHERE** clause is applied:
Table 4.5 Intermediate unification showing the union of Table 4.3 and Table 4.3

<table>
<thead>
<tr>
<th>Intermediate Table 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Str$</td>
</tr>
<tr>
<td>&quot;write stuff&quot;</td>
</tr>
</tbody>
</table>

Now that each of the unification tables have been joined along their syntactic variables the predicates are applied to each row. After applying the WHERE, only one row within the table remains. Because the table isn’t empty it means that this match is to be included in the matches returned by a srcQL query.

4.4 Grammar

The srcQL grammar is simple in that most of the expressions are simply separated by the clause keywords and everything in between is considered to be part of an XPath or srcPat expression depending on the clauses behavior.

```
srcQL ::= srcQLQuery | srcPatOnly;
srcQLQuery ::= selectClause *(containsClause | whereClause) [orderingClauses];
srcPatOnly ::= !("SELECT") *anytoken;
selectClause ::= "SELECT" ["SrcPat"] +anyNonClause [withinClause];
containsClause ::= "CONTAINS" ["XPath"] +anyNonClause [withinClause]
  followedByClause;
followedByClause ::= "FOLLOWED BY" ["XPath"] +anyNonClause [withinClause];
withinClause ::= ["SrcPat"] +anyNonClause;
whereClause ::= "WHERE" xorExpr;
xorExpr ::= orExpr *("XOR" orExpr);
orExpr ::= andExpr *("OR" andExpr);
```
andExpr ::= equalityComparison *("AND" equalityComparison);
equalityComparison ::= operandExp [("==" | "/=" | "/<" | ">" | ">="
    | "/=") operandExpr];
operandExpr ::= "(" xorExpr ")" | floatingPointNumer | integer |
    stringLiteral | syntacticVariable | functionCall;
orderingClauses ::= syntacticCategoryGrouping | ([totalOrderingClause]
    [ groupByClause [ groupOrderingCause ] ]); 
syntacticCategoryGrouping ::= "GROUP BY" "SyntacticCategories";
totalOrderingClause ::= "ORDER BY" (functionCall | syntacticVariable);
groupByClause ::= "GROUP BY" functionCall | syntacticVariable;
groupOrderingCause ::= "ORDER BY" functionCall;

Figure 4.2 EBNF grammar for srcQL.
CHAPTER 5

Implementation of & Using srcQL

In this section we describe how the implementation of srcPat and srcQL is done, how they integrate the tools mentioned in CHAPTER 2, and how complex parts of the language are performed such as unification. In 5.2 we present examples of how to use srcQL on sample C++ source code.

5.1 Implementation

The implementation of srcQL is broken into two major parts, first, is srcPat and second, is the srcQL operators. srcPat is responsible for processing user provided patterns and converting them to XPath. Because the srcQL operators provide a means of specifying relationships between multiple patterns the second step is sewing the srcPat patterns into a single XPath expression.

srcQL’s implementation is broken down into several different steps that consist of compiling the initial query into a srcQL AST and creating an XPath AST from the different srcPat expressions and srcQL operators. The steps for creating the XPath AST from the srcQL AST are as follows:

1. **Compile srcQL**: Compile the srcQL expression into its AST.

2. **Traverse the srcQL AST in Depth-First-Order**: During the traversal of the srcQL tree each of the srcPath expressions is processed into the XPath AST,
this is described later. Upon the visitation of a srcQL operator, its contents are processed into an XPath as either a srcPat express or user provided XPath.

3. **Compilation of Predicate expressions:** After the traversal used to build the XPath query, the predicates are compiled into delegate functions for use during the query’s execution.

The implementation of srcPat involves the conversion of a srcPat pattern into XPath. In theory, one could write any srcPat expression directly in XPath. However, in practice this would be very difficult as the XPath expression necessary for all but trivial matches are extremely long and complicated. The conversion of srcPat to XPath can be described in the following steps:

1. **Preprocess query:** Take the source code provided by the user and record all syntactic variables within it, update the syntactic variables so that they can be processed by srcML.

2. **Run srcML on query:** Run srcML on the preprocessed query.

3. **Check for special macro case:** Check after the query is processed check to see if the provided query has macro markup. Because macros and function calls often look similar, the lack of a semi-colon in a pattern can mark the function as a macro. We handle this by re-adding a semi-colon to the end of the user’s expression rerunning srcML on the expression and then correctly selecting the context of the expression based upon the new markup, but instead of selecting the expression statement element the
expression element is chosen as the starting element for XPath construction. In the event that we encounter this case step 4 is skipped.

4. **Pattern context selection:** The starting element of the pattern is selected from within the srcML. The element selection is based upon an existing list of known acceptable search contexts.

5. **Process into XPath:** Starting at the selected search context element a document order traversal of the current element and all siblings of that element. This during this process the XML markup is turned into an XPath representation that allows for the removal of the syntactic variable elements and creates the XPath necessary for selecting elements of the syntactic variable. The pattern for selecting elements of a syntactic variable is used during unification to locate their values.

The combination of multiple srcQL operators is done using a tree rewriting system which knows how to correctly add different operators into the AST which represents the XPath that is being compiled. A heuristic is used to select where within a pattern an operator is added. The heuristic simply locates the first node within the related srcQL AST and adds the operators XPath query as the last child of that element.

srcQL is written in C# using ANTLR [Parr] and XPath with a custom XSLT context which provides additional XPath extension functions. The XSLT context and the XPath functions are never exposed to the user through the language.
User provided XPath expressions are treated as strings by srcQL and are not checked for correctness of any kind during srcQL compilation. This means that it is usually best to stick to a single path or tag name instead of trying to build complicated expressions using only XPath.

The goal of srcPat is to hide this complexity so that users may use syntax of programming language they are using instead of having to write or reason about complicated XPath or srcML’s markup. srcPat generates the XPath, the srcML markup, and the syntactic variables within the expression.

5.2 Using srcQL to Query C++

The following source code is used in several of the following examples to demonstrate how to use srcQL and srcPat to preform queries on source code. We use the source code in Figure 5.1 to demonstrate a number of srcQL queries.

The first example of a query is to locate each expression that uses a specific operator. For example, to retrieve all of the expressions that contain the operator new, the srcQL would be

\[ \text{new } $T \]

When applied to the source code in Figure 5.1, the result would be to match all uses of the new operator, including those of the form: `new bar()`, `new bar2()`, `new bar3()`, `new bar4`, and `new bar5[q]`. The syntactic matching of $T is to the operand of the new expression.
The example query “locate all of the if statements within a program” can be done as follows.

```
SELECT srcPat if() { }
```

1. void foo1(bar* x) {
2.     bar* x = new bar();
3.     (*x).open();
4.     delete i;
5. }
6.
7. void foo2(bar x) {
8.     bar* a = new bar2();
9.     x.open();
10.    delete i;
11. }
12.
13. void foo3(bar* x) {
14.     barSuperClass* i = new bar3();
15.     if(i) {
16.         i->open();
17.     }
18.    delete i;
19. }
20.
21. void foo4(bar x) {
22.     x.open(new bar4);
23.     delete i;
24. }
25.
26. class myClass {
27.     void foo5(bar x) {
28.         delete i;
29.         x.open();
30.         if(x) {
31.             x.push(new bar5[q]);
32.         }else{
33.             x.pop();
34.         }
35. }
36. ```
The absences of both the conditional and body of the if-statement allows for any if statement to be located with any conditional and any block. This query will also locate if-statements without a following block because the block is implicitly part of the syntax of the C++ programming language and the brackets are omitted from the search pattern. The query would match the following:

```
1. if(i) {
    i->open();
}

2. if(x) {
    x.push(new bar5[q]);
} else {
    x.pop();
}
```

The query “Show me all of the expression statements which contain a call to new.” can be achieved by specifying the search context for the query using SELECT:

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

For the code in Figure 3.2, the results would be:
srcQL provides additional types of queries which allow for more advanced query
selections such as locating all of the types that are used with operator new. srcQL’s
SELECT provides a means for selecting syntactic variables from within expressions, so
the srcQL query would look the following:

SELECT $T CONTAINS new $T

1. bar()
2. bar2()
3. bar3()
4. bar4
5. bar5[q]

“Give me all of the pointer declarations which are assigned at declaration with a
call to new using the same type that was used with the pointers declaration.” srcPat
provides unification which can easily handle queries such as this.

$T* $Var = new $T

1. bar* x = new bar()

“Get all of the void function definitions which occur within classes.”

SELECT srcPat void $Func() { }
WITHIN srcPat class $ClassName { };
1. void foo5(bar x) {
    delete i;
    x.open();
    if(x) {
        x.push(new bar5[q]);
    }else{
        x.pop();
    }
}

“Locate all of the declarations of pointers where the name of the type matches a
regular expression.”

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

1. barSuperClass* i = new bar3();

“Find all of the functions, in which, a variable is declared and initialized with new,
then, following that, the open function is called on that same variable inside of an if
statement where that same variable is checked, and after that is deleted.” The srcQL
unification allows for syntactic variables and be constrained.

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

FOLLOWED BY delete $I

1. void foo3(bar* x) { 
   
   barSuperClass* i = new bar3();

   if(i) {
      i->open();
   }

   delete i;
}

“Get all of declarations of pointers and sort them alphabetically by variable name.”

SELECT(srcPat) $T* $I

ORDER BY $I

1. bar* a = new bar2()

2. barSuperClass* i = new bar3()

3. bar* x = new bar()

“Show me all of the syntactic categories in which operator new is used.” The syntactic categories are given as srcML element names.

SELECT(srcPat) new $T

ORDER BY SyntacticCategories

dcl_stmt/decl/init

1. new bar()
2. new bar2()
3. new bar3()

expr_stmt/expr/call/argument_list/argument/expr

1. new bar4
2. new bar5[q]

5.3 Limitations

We now discuss some of the limitations of srcQL. Many of the limitations arose near the end of the development process and will be addressed in future versions of the language. One of the current limitations of srcQL is that the entire DOM must be loaded into memory in order to perform unification and other analysis that can not easily be done on the fly. This can be a limiting factor when it comes to performing queries on a very large system. We have tested srcQL on GCC, which is a system of non-trivial size, however this is not the upper limit of srcQL. The need for the entire XML DOM to be stored in memory will be addressed in a future version of the language and thus srcQL will be extended to systems of any size.

A limitation of the language itself is its inability to nest or combine multiple queries into a single query and perform set operations upon results of multiple queries. The nesting and combination of queries would allow for a much more advanced complex set of queries giving the programmer a fine grain level control over exact matches within the source code. Set operations such as union, intersection and set minus are advanced operations. Both the nesting and set operations would provide a means of expressiveness
which is not currently part of the srcQL language. We discuss these additions in more
detail in CHAPTER 7.
CHAPTER 6

Related Work

Few languages have been created for the purpose of source code querying. Those that have been created normally require the user to have intimate knowledge of some form of internal tree structure, such as an AST, or the language grammar. If they implement a language explicitly for querying, the user must also learn a new language that, based on studies of previously created tools, do not follow typical query-language structure and take time to understand. They also require compilation of the source code.

We have studied tools that are used for Program Transformation. This is primarily because Program Transformation requires some sort of matching to specify the location for transformation. For the program transformation languages, we assume that minimally the technique used to locate transformation sites could also be used as a query language. Here are some of the languages we showed: JQuery[Janzen and Volder 2003], PMD[Dangel], SCRUPLE[Paul 1992], TAWK[Atkinson and Griswold 2006], CodeQuest[Hajiyev, Verbaere et al. 2005], Aria[Devanbu, Rosenblum et al. 1996], A*[Ladd and Ramming 1995], LINQ\(^2\), XQuery\(^3\) and srcML.


\(^3\) http://www.w3.org/TR/xpath-datamodel/
TXL[Cordey, Dean et al. 2001], RASCAL[Klint, van der Storm et al. 2009], DMS[Baxter, Pidgeon et al. 2004], ELAN[Borovansky, Kirchner et al. 1996], Stratego[Visser 2001], and Coccinelle[Padioleau, Lawall et al. 2008] are all languages that perform Program Transformations. With the exception of Coccinelle, they require full compilation and have the capability to, minimally, match specific sections of code specified by the user. This can be considered a form of querying even if, with these tools, the goal is the transformation, not the query. Coccinelle uses variables to substitute isomorphisms in source code. To find the location of these isomorphisms, it uses a model checking algorithm called CTL. Its application and srcPat only overlap in terms of applicability to the issue of program transformation. RASCAL defines an imperative programming syntax to specify queries and transformation. Its language works on an AST. DMS, STRATEGO, and ELAN are all similar in that they define matching and transformation in terms of an AST. TXL defines a functional syntax for the same reason as RASCAL and works at the level of language grammar. The above tools, while able to perform some querying, are difficult to use for the purpose of querying. The functional, imperative, and rule-based languages they employ are hard to use for describing queries. srcQL uses the syntax of the language it’s working on to describe queries, which allows the programmer to work with readily available knowledge instead of having to understand low-level concepts like language grammars.
SOUL [Roover, Noguera et al. 2011], and RScript [Klint 2003] provide functional languages in order to do matching and transformation. Browse-By-Query [MacDonald 2011] (BBQ) [MacDonald 2011] provides a language for querying source code that’s easy to learn. However, it does not provide an easy means of specifying patterns because one has to have initial knowledge of the source code in order to perform queries upon it and there is lack of documentation about the language. SOUL, RScript, SemmleCode [Verbaere, Hajiyev et al. 2007], QL [Moor, Sereni et al. 2008], Java Tools Language (JTL) [Cohen, Gil et al. 2006], JRelCal [Rademaker 2008], and CrocoPat [Beyer and Lewerentz 2003] 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 [Kniesel and Bardey 2006] performs querying in order to provide aspect weaving into a Java program, and as such requires the AST in order perform queries.

Functional languages are not a natural way to specify source code queries. The tools mentioned above are not all explicitly for source code querying but their users do have to accomplish some sort of matching. srcQL can remove the need to force users to learn complicated grammar or expression matching techniques and, instead, use the easy-to-learn format of srcQL expressions. This is what makes srcQL a great tool for querying.
XPath, LINQ and XQuery are the most widely used tools when it comes to querying the XML structure of srcML, all of which were studied thoroughly. XPath is the most widely used XML querying tool. XPath provides a simple language to query XML, however, the downside to it is that it can require experts in XPath to write more complex queries. All but trivial queries on the srcML XML structure require the expert level knowledge and as such most developers shy away from using XPath. What XPath lacks in readability, it makes up for in speed. Execution of an XPath query is extremely fast compared to other querying tools we have studied; this is why it was chosen as the underlying language to implement srcQL and srcPat.

LINQ is extremely easy to write and allows for the maximum amount of expressiveness and readability when querying the srcML XML structure. However, what the LINQ language gains in readability it loses in speed. This is because LINQ is executed from the language it was written in and, as such, is unable to make more complex optimizations that are available to tools such as XPath. While LINQ provides the necessary extensions and syntax for compiling and executing queries on source code the degradation in speed makes it too slow for the implementation of srcQL and srcPat.

XQuery is currently not widely supported by many of the existing XML libraries, but the language syntax and behavior would have made it better for specifying queries for the srcML structure. Due to the lack of support for XQuery it was excluded from consideration as a possible implementation language.
The main downside to all three of the languages, XPath, LINQ and XQuery is that they only allow the user to operate at the XML level of the source code. This prevents the user from simply writing queries in terms of source code which is what they would usually do.

Another tool used for ad-hoc queries on source code is grep. Grep and tools like it are great for searching for regular expression patterns within arbitrary text documents; however, they have their limitations. Using a regular expression search only considers the text of a document and not the internal structure of source code. This makes it impossible to locate some segments of source code without manual inspection of results.
CHAPTER 7
Conclusions & Future Work

This thesis presented the design and implementation of a novel syntax aware query language for C++. The language srcQL is built on top of a syntactic aware pattern matching language, srcPat. The language is compiled into XPath and then applied to srcML (XML) representations of source code programs. srcQL was demonstrated on a number of examples and appears to be usable and natural for developers to formulate queries. The language is also fairly scalable and efficient.

There are still several limitations on srcQL; the performance, amount of code being queried, matching capability and expressiveness. We will address each of those issues in the following way.

The limitations on performance and the amount of code being queried limitations are due to two issues; 1) the amount of time it takes to load the entire DOM into memory, and 2) the time it takes to compute unification after all results have been gathered.

The time it takes to load the DOM into memory can be removed as a performance limitation if, instead of loading the entire DOM, only a single translation unit is loaded at a time. This is could greatly increase performance because the amount of memory needed would be much smaller than the entire DOM but the trade-off would be that any
time there is a cross reference between two files the entire document would have to be reparsed.

The time it takes to compute unification could be reduced in one of two ways; first, if the table joins could be replaced with something else such as being directly coded into XPath this would allow for much faster evaluation of unification. Second, if the unification algorithm wasn’t written in C# than execution time could be cut drastically.

Matching capability and expressiveness go hand in hand. The limitations of the srcQL matching process, while few, do exist. The limitations are generally the result of the heuristics that are used while generating the XPath of a srcPat and the unification. The issues that arise from heuristic of a srcPat XPath are usually related to duplicity, because the language currently lacks a way to express syntactic sequences of elements such as parameter lists without limiting them to N or more arguments. Both the matching capabilities expressiveness can be improved by providing additions to the language in the form of set operations. Most of the matching limitations can be expressed as a set operation on two sets, such as a union, or set minus. Having the additional set operations would also improve the expressiveness of the language because it would allow for the number of arguments to be limited to a single number via set subtraction operation.

The main limitation of unification within srcQL is that it is done naively using a basic string comparison. This could be fixed by coming up with a more comprehensive algorithm which would improve the number of false positives and false negatives.
matches. The current algorithm doesn’t handle nested expressions at all and doesn’t allow more general statements to be matched.

### 7.1 Future Work

This work has served to foster improvements in future languages used for querying and modifying source code. Our plans for the future of the srcQL language are to add operations to provide extra capabilities to developers. The features we have planned for the future include; filtering, checking if a syntactic variable or match context exists within a previous query and set operations. These extensions would allow for maximum amount of flexibility and capability of the srcQL language.

For the filtering and exists within query operators would be available using a `FROM` operator. `FROM` could be used in two different ways:

*FROM* could be used as part of a nested query in order to allow for filtering through multiple expression or selection of a syntactic variable from a complex srcQL expression. For example, selecting all of the function definitions from within all classes that have a factory function, could done using the following example,

```sql
SELECT src:function
FROM SELECT src:class
   CONTAINS $X factory();
```
FROM could also be used to check if part of a query exists as part of a previous query so as to reject or accept a match based on a previous search. For example, selecting all of the classes which inherit from a class within a particular namespace:

```
FROM relopsClsNames
SELECT $T
FROM SELECT srcPat class $T { };
WITHIN srcPat namespace relops {}
SELECT srcPat class $CLSNAME: $X { };
WHERE $X in relopsClsNames
```

The above query would allow a previous set to be given and named so that it could be used again to check if $X is a match to any of the class names within the relops namespace.

The set operations would be similar to those described within using the FROM clause. The set operations would include union, intersection and set minus. Each of these operations would be binary operators which can be used to represent things such as logical OR, AND and XOR operations on sets of results.

### 7.2 Applications of srcQL

srcQL’s application to software engineering problems is broad. Presented here are some of the problems it could address and how it solves them.
7.2.1 Adaptive Maintenance

There are many examples of adaptive maintenance that srcQL can help developers accomplish. One such task is Program Transformation. Program Transformation can be augmented with the use of srcQL using the concept of a Syntactic Category. A Syntactic Category defines different syntactic formats of semantically similar expressions within source code. 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 them. For example,

```c++
int *ptr = new int();

Foo(new int());
```

shows two variant uses of operator `new`. One is in a declaration expression and the other is in a function call. Should a developer want to write a transformation that will affect all operator new calls, they would need to know what variations there are in the uses of operator `new`.

Using srcQL, a developer would be able to find every version of operator new calls and how many times each of these variations occurs. This allows developers to write a transformation for each separate variation or, perhaps, decide that it’s more time-efficient to manually rewrite the code. The other option would be to look through many millions of lines of code manually. srcQL could also be used to pinpoint these variations
in a srcML archive programmatically, allowing transformations to be applied directly to the result of a srcQL query.

Refactoring is another adaptive maintenance task that srcQL supports naturally. Segments of code to be refactored can be easily found. As with the transformations described above, srcQL’s API can be used to return locations to these points in the program. Because of this, they may be programmatically refactored or manually refactored.

7.2.2 Corrective Maintenance

When a bug is discovered, developers must often find areas in the code which must be modified. Not only this, but developers must also analyze the effects a bug fix may have on other parts of the system. srcQL is able to support such efforts by making it easier to find the location of statements containing faulty code, which allows the developer to now make decisions on how the code should be corrected. Since srcQL can be made to return results throughout the system, the impact of the bug fix can more easily be measured and taken into account by developers. Again, the primary advantage is the ease and quickness with which queries may be created and performed.

7.2.3 Program Comprehension and Understanding

The primary concern of Program Comprehension is to allow the developer to understand parts of a large code base for the purpose of maintenance. srcQL’s ability to
find segments of code with similar structure and keywords between multiple modules; files, classes, etc., allows developers to both query for these patterns quickly and without the effort of trudging through difficult and unfamiliar query syntax. This would not only be advantageous for new programmers that are not yet experienced with the system they’re working on, but for veteran programmers who want to ensure that they’re addressing every related segment of code without having to spend the large amounts of time attempting to double-check their work.
REFERENCES


[Visser, E. 2001]. Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5. Rewriting techniques and applications, Springer.