Towards Query Formulation and Visualization of Structural Search Results

Oleksandr Panchenko
Hasso Plattner Institute for Software Systems Engineering
P.O. Box 900460, 14440 Potsdam, Germany
panchenko@hpi.uni-potsdam.de

Arian Treffer
Hasso Plattner Institute for Software Systems Engineering
P.O. Box 900460, 14440 Potsdam, Germany
arian.treffer@student.hpi.uni-potsdam.de

Alexander Zeier
Hasso Plattner Institute for Software Systems Engineering
P.O. Box 900460, 14440 Potsdam, Germany
zeier@hpi.uni-potsdam.de

ABSTRACT

Source code search goes far beyond simple textual search. One possibility of improving code search is the utilization of structural information in form of abstract syntax trees (ASTs). However, developers usually work with the textual representation of source code and, thus, have difficulties in expressing their queries as fragments of abstract syntax trees and in interpreting the results. This paper addresses assistance of query composition and search result visualization. Query formulation is considered to be an iterative process. After one query is run, the AST vertices neighbored to the result vertices are analyzed to propose refinement options for the next query. Search results are visualized in a tree view which aggregates all matches in a compact way instead of showing a small number of ranked matches.

Categories and Subject Descriptors

D.2.6.e [**Software Engineering**]: Design Tools and Techniques—*Programmer workbench*

General Terms

Source code search, Abstract syntax trees, XPath, Search results visualization

Keywords

Program analysis, Visual programming and program visualization

1. INTRODUCTION

Source code documents are of dual nature: they are in fact texts containing information for developers and they have explicit structure for compilers and other tools. Several structural representations of source code exist: abstract syntax tree (AST), call graph, data flow graph, and others. Until recently, traditional search engines have been used to search in large source code repositories. These search engines use the textual representation and disregard structural characteristics of code. Recent efforts added some metadata [3], rough structural information [5, 7], and source code structure in form of ASTs [11] to the search index. Nevertheless, existing repositories and query languages lack support for query formulation and results representation. This paper aims at bridging this gap.

This paper proposes to use widely accepted XML querying languages for querying source code structures represented as ASTs. As any XML document object model, an AST is in fact a tree. Four simplified examples of ASTs are depictured in Figure 1. An AST is a detailed tree-based representation of the syntactic structure of a source code document. Each vertex of the tree represents a source code entity and belongs to one of 6 categories: identifier (idf), literal (lit), statement (s), clause (v), operator (opt), or compound (c). AST vertex categories are presented in Figure 1 as namespaces. An edge represents a containment relation between entities. Many questions about the structure of source code can be answered by analyzing ASTs. For example, the queries "find all reading accesses to a certain database table" or "find all method invocations which can be critical for performance because of passing data by value" are in essence structural patterns (AST fragments) to be found. Such structural queries can be formulated as XPath expressions. The obvious advantage of XPath over keyword-based search and *grep* is the ability to expresses fine-grained relations between source code entities. The database schema and the implementation of an XPath query engine for ASTs was discussed in our previous work [11].

Since developers usually work with the textual representation of source code, they have difficulties in formulating XPath queries for AST and in interpreting the result matches, which are AST fragments. This paper addresses these challenges and proposes an approach for XPath query formulation based on auto-completion and iterative query refinement. AST vertices, which are neighbored to the result matches of a query, are analyzed, grouped by their type, and used to generate refinement advice for the next query. Moreover, an appropriate result visualization makes searching more efficient.

The examples shown in this paper originate from an en-

SUITE '10, May 1 2010, Cape Town, South Africa

terprise system SAP Business Suite[1]. The language of this system is ABAP[2], which is a fourth-generation programming language for data processing in commercial applications. In addition to language elements which are usual for programming languages, ABAP offers functions for background task scheduling, authority checks, operations with database tables, arrays, files, user interface elements, strings, date and time, etc., as language elements. For example, access of a database table is represented by the *SELECT* statement, and reset of a variable to its initial value is provided by the *CLEAR* statement. Therefore, the language is very rich with more than 800 non-reserved keywords and more than 1100 types of AST vertices.

Eight ABAP developers were interviewed to detect their needs in context of code searching. All of them use search mostly while conducting code reviews, quality checks and searching for API usage examples. Many of the queries conducted by the developers within these scenarios consist of identifiers and syntactical patterns. Currently, the developers use a search engine which is fast but accepts only simple textual queries; a where-used list for identifiers which is slow and does not allow distinguishing between different types of use; and a set of pre-defined checks for violations of program structure.

2. RELATED WORK

Holmes et al. used a relational database to store source code and to query it to recommend relevant examples [5]. Nevertheless, this repository stores source code with a coarse level of detail.

Begel proposed enriching terms stored in an index with metadata about the role of the term, type of usage, etc. [3]. A similar approach is used by existing code search engines, e.g., Codase, Google Code Search, Koders, Krugle, and Merobase. Stored metadata does not contain information about relations between fine-grained entities and, therefore, the query interface is simple.

Linstead et al. used a relational data model to store data with a middle level of detail. Information about more fine-grained characteristics is precomputed and aggregated into fingerprints [7]. To answer new queries, new fingerprints should be implemented.

JQuery is a customizable query-based viewer of source code [10]. Its query language operates on high-level facts extracted from ASTs.

Chew used a query language for syntax-level analysis of C/C++ programs based on AST [4].

Serialization of source code in XML format has been discussed several times. JavaML is a markup language for referencing source code entities, transforming, and querying on a higher level of abstraction [1]. srcML provides explicit markup of syntactic information within source code to support various program analysis, fact extraction, and reverse engineering tasks [8]. XSDML uses an XML representation of source code for manipulation and refactoring [9]. Nevertheless, until now XPath has only been used to address AST vertices or to sequentially process a list of source code documents. This paper reports on the usage of XPath for the querying of many documents in a source code repository simultaneously. Because of a large number of ASTs in the repository, and potentially large number of results, current methods for query formulation and result visualization are not optimal.

Although some effort has been spent on investigation of the ranking methods for source code search [6, 7], search results are still represented as a simple list.

Generally, code search engines have a simple interface but are used for rough search. Source code query languages are more expressive, but have complex semantic and are difficult to use. Although many code query languages exist, their focus has been on a search for syntactic bugs and other questionable constructs. In these scenarios it is acceptable to require a high effort to formulate a query because the number of such queries is limited and they can be prepared in advance. However, in order to integrate fine-grained structural search more closely into daily development activities, the query formulation should be effortless.

3. SEARCH ASSISTANCE

Users of a search engine work iteratively, entering queries, analyzing the results and refining the previous query. A very obvious start is a simple keyword-based query. Whereas developer defined strings (class names, method names, parameters, variables, etc.) are placed as leaves of the AST, other vertices define the roles of the leaves and their interplay or context. Therefore, the matches of a keyword-based search are AST leaves. In contrast to the traditional search engines, our repository contains a complete AST structure and therefore the role of the match and its context are easily obtained at runtime.

The following scenario illustrates the approach. A developer wants to find a location where a database table named FILE is read. He starts with a simple keyword-based query:

$$\text{“}FILE\text{”} \tag{1}$$

In background, this query is translated to the following XPath query:

$$//idf : FILE \tag{2}$$

Several matches have been found (Figure 1, Action A1). An analysis of the found ASTs provides a list of all ancestors of the matches. This list is used to identify the role of each match and its context. In the example given in Figure 1, Action A2, the matches are used: (A) on the left-hand side of an assignment statement and in a *CLEAR* statement; (B) as a container for results of a *SELECT* statement; (C) as a parameter value passed to a method, whereby the method call is inside a loop; and (D) in a *WHERE* clause of a *SELECT* statement. These roles are aggregated by type and proposed to the developer as a list of query refinements:

$$//s : COMPUTE//idf : FILE \tag{3}$$
$$//s : CLEAR//idf : FILE \tag{4}$$
$$//s : SELECT//idf : FILE \tag{5}$$
$$//s : CALL\_METHOD//idf : FILE \tag{6}$$
$$//s : LOOP//idf : FILE \tag{7}$$

In our interviews with developers, we discovered that statements are typical targets of search while conducting code reviews or searches for examples. Therefore, the first list of proposals is restricted to statements. The proposals can be presented as a list of alternatives if the developer starts

[1]http://www.sap.com/solutions/business-suite/
[2]http://www.sdn.sap.com/irj/sdn/abap/

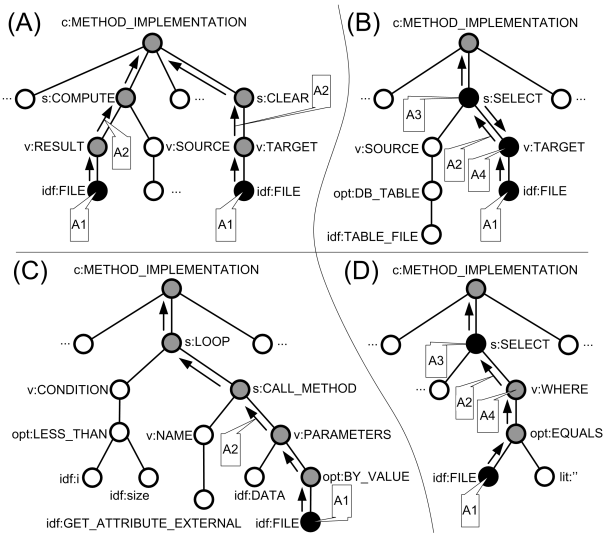

Figure 1: Four AST Examples

with an identifier or literal (usually placed in the right-most part of the expression) and then adds some conditions at the beginning of the expression. In this case, unnatural move of the cursor from the right to the left can be avoided. For top-down query formulation, if the developer starts with a statement and specifies the identifier later, it is natural to provide help while-you-type in the auto-complete function (Figure 2). The categories of AST vertices are automatically proposed, but can be corrected if necessary.

The developer decides to investigate all *SELECT* statements and accepts the proposal #5 (Figure 1, Action A3). Nevertheless, several possibilities still exist. The identifier FILE stands not only for a container for results of the *SELECT* statement (Figure 1.B, Action A4), but also for a field of a table (Figure 1.D, Action A4). The system analyzes possible ways of refinement based on data in the repository and makes the following proposals:

$$//s:SELECT/v:TARGET/idf:FILE \quad (8)$$
$$//s:SELECT/v:WHERE//idf:FILE \quad (9)$$

By selecting option #8 the developer finishes the query formulation. Since the names of AST vertices can differ from the corresponding keywords in source code, an example of source code snippet is shown for each option if necessary. This example is retrieved from the repository at the runtime. The auto-complete feature for XPath is used quite broadly and is implemented in numerous tools, e.g., in Altova XMLSpy[3]. However, these implementations rely on XML schema which is too complex for fourth-generation programming languages. Therefore, our prototype uses data in the database for the advice generation. Moreover, in contrast to the XML schema-based approaches, the database analysis identifies the most used patterns and can sort advice by frequency. Other possibilities of sorting include alphabetic sorting and sorting by frequency of usage in search queries. Our interviewees were not unanimous at this point, thus, we left it configurable.

[3]http://www.altova.com/xmlspy.html

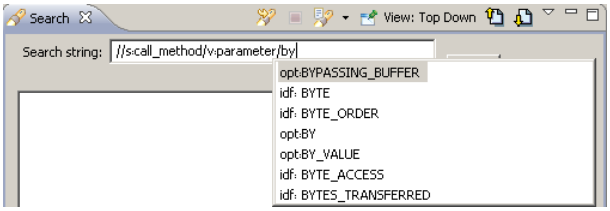

Figure 2: Auto Complete Function

4. RESULT VISUALIZATION

Most search engines operate on documents. Thus, the result of the search is a set of documents with each document containing one or more matches for the search expression. Source code search is no exception to this rule. Therefore, one part of the result view shows the list of result documents (left part of Figure 3.A), including the program name, a content snippet and some metadata.

One aspect of visualization is the ranking of the result list. The combination of structural and textual information can improve the ranking. If the search query contains more than one term, the result hits can be sorted by the distance between those terms in the result documents. This sorting criteria works well for natural language documents, but in software documents two terms that are placed close to each other may not be related or even belong to different namespaces. This feature of source code is taken into account by calculating the tree distance between the corresponding vertices of the AST.

The role of the keyword also is derived from the AST. Using this information, matches in class or method names can be ranked higher than matches in local variable names.

Another metric for ranking documents is the page rank. The page rank indicates how often a document is referenced by other documents. It has been shown that the page rank is applicable to software documents too [6, 7].

However, in many cases the result is large because the query is not selective enough. Many of our interviewees start with a simple query instead of spending time for formulation of a precise query which probably is too restrictive. In this case, simply showing the result list, even if sorted, is not sufficient. Instead of looking at the first matches, the user needs to further restrict the search result.

Here we take advantage of the structural nature of the result data to provide an overview over a large number of matches. Additionally, each match is provided with a context that is more expressive and formal than a simple content snippet.

After the processing of the XPath query, not only the result vertices themselves, but also all of their ascendants are returned by the database. Thus, the search result is a set of sub-trees. Example sub-trees are presented in Figure 1 as chains of dark-grey AST vertices. In these sub-trees all leaves are matches of the search and all other vertices are ascendants of one or more matches.

To provide a better overview over this set of sub-trees, all sub-trees are aggregated into one result tree. Starting with the roots, vertices of the same type are joined into one vertex of the final result tree. This view is called a top-down result tree and is shown on the right-hand side in Figure 3.A. The number of matches indicates how many sub-trees were aggregated. This view provides a high-level overview of all

matches grouped by their role and usage context with the possibility of drilling down to single matches. The chain of AST vertices between the root of the result tree and the match corresponds to the chain in the original AST. For example, here the developer sees that identifier FILE was used in data definition fragments 42 times (25 times as a variable name and 17 times as a type name), variable FILE was reset 3 times and referenced inside a loop 77 times, etc.

Since developers are usually interested in local context (few nearest ascendants) of the match, we propose additional layout where the sub-trees are inverted and vertices are joined starting from the leaves (Figure 3.B). This view allows exploring the result tree starting with the role of the match and its local context.

The result view provides a compact yet detailed overview of *all* matches and their parents, and also supports the iterative process of query formulation. In both tree views, the developer double-click on an AST vertex to add it to the query or right-click on the vertex to see the list of documents containing this match. It is possible to directly jump from here to the location in the source code.

Another popular way of visualizing hierarchal data is a treemap [2], where each vertex is shown as a rectangle. Child vertices are arranged in a way that they fill the entire rectangle of their parent vertex. It is possible to show the entire tree at once or just a few topmost layers. Then, by clicking on a vertex, the user can expand parts of the tree. Figure 3.C shows the *METHOD_IMPLEMENTATION* sub-tree that is part of the top-down result tree in Figure 3.A. The size indicates the number of result documents and the color shows the number of matches.

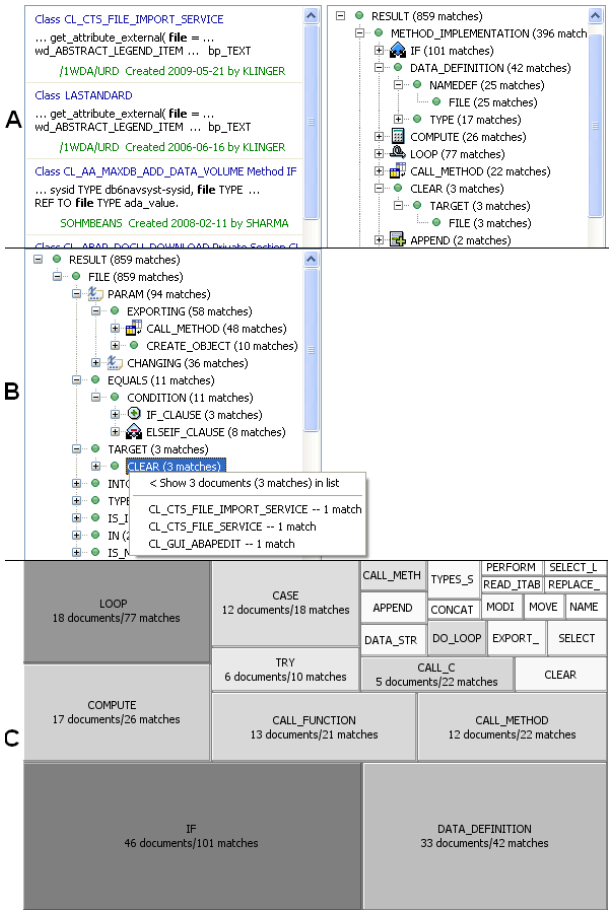

Figure 3: Visualizations of Query#2 Result

5. CONCLUSION

This paper presents an approach for assisting developers in query formulation and search result visualization. The application of this approach will enable effortless and interactive source code searching for fine-grained syntactic patterns. With some adaptation, these principles can be applied to other source code query languages than XPath.

6. REFERENCES

[1] G. J. Badros. JavaML: a markup language for Java source code. In *Proceedings of the 9th international World Wide Web conference on Computer networks*, pages 159–177, 2000.

[2] M. Balzer, O. Deussen, and C. Lewerentz. Voronoi treemaps for the visualization of software metrics. In *Proceedings of the Symposium on Software Visualization*, pages 165–172. ACM, 2005.

[3] A. Begel. Codifier: A programmer-centric search user interface. In *Proceedings of the Workshop on Human-Computer Interaction and Information Retrieval*, pages 23–24, 2007.

[4] R. F. Crew. ASTLOG: a language for examining abstract syntax trees. In *Proceedings of the Conference on Domain-Specific Languages*, pages 18–18, Berkeley, CA, USA, 1997. USENIX Association.

[5] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate Structural Context Matching: An Approach to Recommend Relevant Examples. *IEEE Trans. on Software Eng.*, 32(12):952–970, 2006.

[6] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto. Ranking Significance of Software Components Based on Use Relations. *IEEE Transactions on Software Eng.*, 31(3):213–225, 2005.

[7] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18(2):300–336, 2008.

[8] J. Maletic, M. Collard, and H. Kagdi. Leveraging XML Technologies in Developing Program Analysis Tools. In *Proceedings of the 4th International Workshop on Adoption-Centric Software Engineering*, pages 80–85, 2004.

[9] K. Maruyama and S. Yamamoto. Design and implementation of an extensible and modifiable refactoring tool. In *Proceedings of the 13th International Workshop on Program Comprehension*, pages 195–204. IEEE Computer Society, 2005.

[10] E. McCormick and K. D. Volder. JQuery: Finding Your Way through Tangled Code. In *Proceedings of the conference on Object-oriented programming systems, languages, and applications*, pages 9–10. ACM, 2004.

[11] O. Panchenko, H. Plattner, and A. Zeier. Efficient Storage and Fast Querying of Source Code. *Information Systems Frontiers, Springer*, 2010.