An Enhanced Graph-based Infrastructure for Software Search Engines

Marcus Schumacher, Colin Atkinson
Chair of Softwareengineering
University of Mannheim, Germany
Email: {schumacher, aktinson}@informatik.uni-mannheim.de

***Abstract*—The first generation of software search engines such as Merobase, Sourcerer etc. showed that it is possible to support reasonably sophisticated searches over large bodies of software components using indices based on full-text search engines (most commonly Lucene). However, the tricks these engines use to map code structure to flat text are not only inflexible, they do not scale well to components composed of multiple program modules (e.g. interfaces, classes etc.) As a result, beyond plain string matching, they are only able to support a limited and a priori fixed set of query types, and are rarely, if ever able, to find components composed of more than one code module. In this paper we present an index representation approach which is able to support the key information bound up in source code in a more accurate, flexible way, and thus efficiently support a much wider range of searches on components composed of multiple program modules.**

I. Introduction

When the first generation of software search engines were constructed, in the 1990s, only a small number of software components were freely available over the Internet. However, with the open source revolution around the turn of the new millennium and the rapid growth of open source repositories [1], as Mili et al. predicted [2] the quantity of software available for search engines to search over dramatically increased. Software researchers were then faced with the challenge of developing a new generation of software search engines, capable of efficiently coping with these large numbers of components. Initially Rational Database Management Systems (RDMSs) were the dominate storage technology used to respond to this challenge. However, when using them to store source code, researchers faced several limitations due to the mismatch between the nature of software in modern programming and the search capabilities of RDMS systems. The latter are optimized for structured SQL queries instead of keyword based searches on unstructured text and do not inherently support a ranking mechanisms for search results.

As a consequence of these problems, Full Text Search Frameworks (FTSF) become the technology of choice for the second generation of software search engines. The most well known and widely used example is the open source framework Lucene, originally developed by Dough Cutting, which is now an official Apache project [3]. All new code search engines that emerged in the 2000s, like OpenHub, Krugle, Sourcerer or Merobase [6] [4] [5] [7] where based on FTSFs, and usually Lucene. However, FTSF based software search engines also have some significant limitations. In particular, their lack of support for relational searches makes it difficult, if not impossible, to search for software components consisting of more than one code module (e.g. class) due to the large variety of dependency relationships that can exist between them.

In this short paper we therefore present a new hybrid approach for architecting software search engines which addresses the problem of dependencies, while still leveraging the capabilities of keyword based searches. This is achieved by combining FTSF technology with graph databases. In the next section we provide more information about the problems encountered in building software search engines, and in the section after that, we discuss the advantages of a hybrid FTSF/graph database solution. In section IV we briefly discuss future work and in section V we conclude with some closing remarks.

II. Code Search Engines

Several of the second generation code search engines that emerged in the early 2000s were discontinued relatively quickly (e.g. Google Code search was shut down in 2012 due to the shortcomings of its search capabilities [9]). Of course, they all offered basic search mechanisms, like keyword based searches over the raw text of source code, but only a few also offer advanced code-oriented search mechanisms such as the ability to search for specific interfaces (e.g. Merobase) or for code where a specific method is invoked (e.g. Krugle). However, it is simply not practical to search for more complex software structures composed of a multitude of parts using index structures in which all code modules are treated as individual, isolated component. To efficiently support more sophisticated searches, more detailed information about the connections (i.e. dependencies) between code modules must be stored. Of course, it is relatively straightforward to store simple information like the name of a superclass, but what about searches where the name of the superclass of the unit where a method is called needs to be identified? Today's search engines are simply not able to support this level of detail in search queries, with the result that a lot of potentially matching components are not found and users still have a lot of manual selection work to do, to identify potentially reusable components that can be used without adaptation.

To date, there are a few software search engines in existence that support limited dependency-aware searches, like Sourcerer or Portfolio. However, they have the major disadvantage that

978-0-7695-5594-2/15 $31.00 © 2015 European Union
DOI 10.1109/MSR.2015.46

complete projects need to be available at parsing time to resolve the dependencies between code modules [10]. However, this information is often not available because there are so many possible ways of configuring the dependencies of a project, like Maven, Ivy, etc. that search engine crawlers need a parser for the configuration files of all existing tools. Also, many projects are not completely available, for example, when code is presented on a web site as a code snippet. As Subramanian and Holmes recognized, such code snippets can be of very high quality because they have been matured and checked over many years of study [11]. Finally, many projects are uploaded to repositories as compiled libraries. In some languages like Java it would be possible to decompile these libraries and index the structure and method signatures of the code modules, but this is not always possible with other programming languages.

The new generation of graph databases have the potential to address many of these issues since they are able to efficiently store structured data encompassing many variable relationships. In the following section we therefore present our proposal for a search engine architecture which, on the one hand supports advanced searches over complex, multi-module components and on the other hand supports the efficient crawling and indexing of decoupled code modules.

III. Infrastructure and Search

A key weakness of the first generation of code search engines based on FTSF frameworks was their limited support for structured queries that exploit the internal structure and architecture of multi-module software components. For example it was not possible to specify in a search that two different keywords have to occur in two different fields with the same field name, which could occur in multi-fields of Lucene. Of course, it is possible to work around this problem by adding additional fields and other artificial structures to the Lucene index, as Hummel [12] did in Merobase to support interface-based searches, but this will only work for a limited number and form of relationships and often gives rise to an unmaintainable set of redundancies.

As described in [8] there have already been numerous attempts to combine a Lucene index with other types of database to combine FTSF search capabilities with relationship-awareness, but these have always been limited by the weaknesses of RDMS databases. The main problem was the huge numbers of "joins" needed to related code modules. Although the number of explicit dependencies between classes are not always that large, classes have often quite a number of implicit dependencies, such as method in-/output parameters or complex types within a method, which collectively give rise to a large number of relationships to other classes. Traditional RDMSs simply do not cope efficiently with a huge number of relationships once the repository reaches a size of millions independent components.

Graph databases do not face this problem, however, because they are explicitly optimized to support data with large numbers of potential relationships. Thus, once an initial node in a data structure has been found, it is possible to traverse very quickly to corresponding neighbor nodes, if desired, by specifying which kind of relationships should be navigated over. As a consequence, graph-based databases can easily cope with the code search scenario because, in general, it is not necessary to visit all neighbour nodes, just the ones which are specified in the search query. To optimally exploit this capability we integrate this graph navigation capability with the rapid node selection capability of FTSFs. More specifically, we introduce a Lucene index to provide more rapid access to the starting nodes in graph based queries. Thus, even for simple keyword based searches, the first step in any search is to perform a search on a Lucene index to find the relevant starting node for the graph-based search. From this node, the search algorithm then collects the detail information about the component from the graph to present to the user.

For all the other possible searches, like interface based search, the chosen search mechanism highly depends on the intention of the user. The Lucene index is therefore carefully structured to efficiently support as many search scenarios as possible. As more complex multi-module search scenarios, e.g. a search which includes requirement for multiple class definitions, the graph also offers a lot of search capabilities out of the box. These are often marginally slower than searches in optimized Lucene indexes, but the difference is negligible. Since certain kinds of searches can be supported in different ways, rules are needed to define which requirements will apply and how the searches will be performed.

A. Database Structure

As already mentioned, the data is not stored just in a Lucene index, as in other search engines, but in a combination of Lucene indexes and graph databases. The specific graph databases we use for our current implementation is Neo4j, one of the leading graph database currently available[13]. Even through Neo4j internally uses its own Lucene indexes to optimize searches, we introduce our own additional indexes in parallel. This is mainly because the indexes used by Neo4j are designed to optimize searches for names of the nodes and their properties, whereas we have slightly different requirements.

The object-oriented programming languages used for software today ensures that source code adheres to a certain set of fundamental rules, albeit in many different ways (e.g. classes have super classes that implement interfaces, methods call methods of other classes, etc). In principle all mainstream object-oriented programming languages have these basic concepts, albeit in a slightly different form with various exceptions, like multiple inheritance which is not possible in every language. Therefore, we have designed the schema of the database to be very general, but on the other hand tailorable to specific contexts.

The central element is the CodeComponent, which is independent of the programming language. This CodeComponent can be of various types, e.g. in Java a class or an interface. These CodeClasses or CodeInterfaces can contain CodeMethods, which can be normal methods, test methods or constructors.

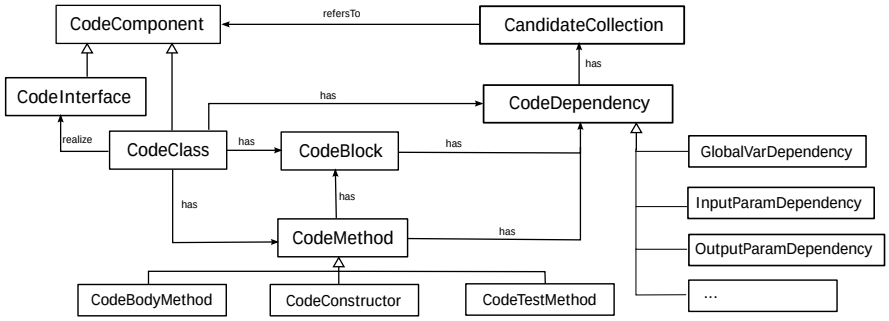

Fig. 1: Snapshot of the Metamodel of the graph database schema

The third important part is the general notion of a block of code - CodeBlock. A CodeBlock may be a method body, but also a static block of code in a class context without any relationship to a method or constructor. These three meta elements share the property that they can have dependencies. These dependencies themselves are in principle also CodeComponents. However, at this point we break the cycle to add another important node, the CandidateCollection, for a variety of reasons. The main reason is to enhance the crawling and parsing process in such a way that it is not necessary at parsing time to have these dependencies available for additional analyzes. However, some basic information about the dependency is usually included in the origin component, such as the import statement and/or the class name.
Since this basic information is not sufficient in a global context to determine with absolute certainty which one is the right dependency we add a dummy node to the CandidateCollection, containing some basic information. In the later crawling and parsing process all real existing classes with this class name and package name are associated with this dummy node. As a result, our parsing process is divided into two steps, the first being the pure parsing of individual files, and the second being the analysis of these dummy nodes to eliminate inappropriate associated classes. To eliminate these associated classes we analyze such things as the invoked methods and their signatures, dependencies and used variables in the case of a superclass. However, we also realize that a precise definition of a valid dependency can only be determined by appropriate tests. Another advantage of this CandidatesCollection node is increased reliability. Especially on the internet, where data is subject to daily fluctuation, it could happen that the data becomes unavailable under the original URL. This may be due to the release of a new version, in which the old component migrates into an archive folder and the version number was part of the URL, or a project changes its hosting (e.g. from Google Projects to GitHub). In this case it would only be a question of time until the new place is re-parsed and the component is re-added to the list of candidates. Of course, the analysis and testing process has to be restarted in this case, but in the meantime another component of the list of candidates can be used instead and in a global context, given the widespread use of version control systems, the probability of a duplicate of a component is relative high.
Another important requirement for this new structure was expandability of all parts of the model. As Fig.1 illustrates, in the case of the methods or dependencies it is possible to define different subtypes of every element. For example, for Input- or OutputParameterDependency it is possible to define other kinds of dependencies, so that specific characteristics of different programming languages can be integrated in the model. This is the reason why we do not show the whole model in this paper, but only the most relevant parts. It also demonstrates why a graph-database provides an optimal basis for software search engines because the schema does not have to be fixed at the beginning. On the contrary, nodes of different types can be added at will at a later time.

B. Search Query Structure

As with all new search technologies, the nature of the query language needs to be enhanced as well. We have therefore also extended the existing MQL (Merobase Query Language) [12] for the new system. The syntax of the previous MQL, which was oriented towards FTSF based indices, requires so-called ”flattened” signatures in non tokenized fields. Thus, for example, a search query to search for a *Calculator* containing two methods, *add* and *sub*, had the following form:

Calculator(add(int;int):int;sub(int;int):int);

This query was split up into three different constraints, the first for the two methods and the second the class name. However, the class name was ranked lower because user are usually more interested in the correct method signatures rather than the name of a class. This also supports similar class names. In the case of the methods the method signature have had to be stored in the old Lucene index in untokenized fields to be sure that exactly this signature is taken into account.
This basic principle is still used in the new query language but with small additions. As already mentioned, to support the old query syntax alongside the new one, instead of only one big index we employ several indexes with a different structure for each of them. One of the indexes supports all the relevant class information, as before, like class name, constructors, contained method signature, etc. This index

makes it possible to continue supporting the old MQL syntax and keep all existing search capabilities. Another index contains information about the content of the methods (e.g. which parameters are used, which other methods are invoked or in which classes a method is contained, as duplicates not only exist at the class level). Finally, we introduce an additional supporting index which contains auxiliary information and is continuously changing. This index contains information about which nodes haven been already indexed, a mapping between Lucene documents and graph database nodes, problems occurring at parsing time, etc. This also supports several scaling mechanisms.
The new type of search query is illustrated by the following example, where a user would like to find components related to a *Car*, which implements an interface *Vehicle* and has capabilities to observer tire pressures.

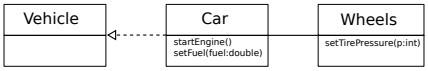

Fig. 2: Structure of the search example

The query for this example in Fig.2 would have the following form:

Defs:(I1:Vehicle(); C1:Car(startEngine(), setFuel(double));
C2:Wheels(setTirePressure(double));
Deps:(I1 ← C1; C1 → C2)

The MQL has been extended by a Defs and a Deps section. The individual signatures can be specified as usual in the Deps section, following the conventions of previous interface based searches. But every signature has to be classified with 'C' or 'I' followed by an Integer, where 'C' stands for a class and 'I' for an interface. This is necessary for the new Deps section in which the individual relations between classes can be defined. At this stage, it is independent of the kind of relationship that exists between the classes. It could be a used global variable, or a method parameter, etc. This is an area for future enhancement as explained in section IV below.
Before the search is performed the sytem checks the query to ensure that the classes from the Deps section are ordered in such a way that the class with the most outgoing relationships will be the first class in the list. After that, the search is performed depending on the complexity of the search query. In a typical case a search for the first class in the list is performed in the main Lucene index containing the information about the class structure. After that, the graph database is traversed by starting at the node of each individual search result to check if the relationship conditions are satisfied as well. If a result contains less methods than expected, an additional search is performed on our method index to find additional components which could be used to extend the origin result component. Here we plan to integrate an adaptation mechanism in the future. If the search process reaches a level of complexity that cannot be managed by a search on Lucene (e.g. because it would require hundreds of individual searches) we ignore the first "Lucene index" step and perform a search directly on the graph database. Nevertheless the Lucene indexes still play an important role, because even in this case, if a search delivers results which do not perfectly match the requirements, a search for every missing constraint can be performed on the Lucene indexes.

IV. Future Work

Since queries can be extremely complex, depending on the scenario, we are planning to integrate a new mechanism to define them. For this purpose, it is advantageous to use an existing tool which most developers are familiar with and which fits perfectly with our MQL syntax - the UML. In particular, UML class diagrams provide information about class and method definitions and their relationships from which queries in our extended MQL syntax can be created. Another extension would be to integrate here also the UML activity or sequence diagrams to extract at this point the process flow within a component to integrate the invoked methods into the search query as well. This will also require a redefinition of the Deps section of the search query to allow the types of relationships between classes to be taken into account. At the moment we just distinguish whether a relationship is between two classes or between a class and an interface. However, then we have to determine whether a variable or a method is used in the related class. Beside these extensions to the MQL query language, we also plan to extend the database with other kinds of information. As every node in the graph database can have individual properties, it would be possible to add information about several metrics, like energy consumption metrics, which would be the useful in mobile development. Alternatively, to support test driven searches and test recommendation systems, additional nodes or properties could be added to hold information about already passed test configurations, as in Janjic [14].

V. Conclusion

Software developers still frequently "reinvent the wheel" and re-implement already existing components from scratch because software search engines are still not sufficiently reliable. Hence, researchers are still searching for better ways to support developer with tools that simplify the task of finding and reusing existing components. However, this is a more complex task than might be expected, because no globally unique identifier can be assigned to software components to rule out the possibility that two components with the same name and same method signatures support different functionality. This new database structure makes it possible for code search engines to support more complex search scenarios and thereby provide more reuse options to software developers.

REFERENCES

[1] A. Deshpande, D. Riehle, The total growth of open source, Fourth Conference on Open Source Systems, Springer Verlag (2008)

[2] Mili, A., Mili, R., Mittermeir, R.: A Survey of Software Reuse Libraries. Annals of Software Engineering 5 (1998)

[3] Hatcher, E., Gospodnetic, O., McCandless, M.: Lucene in Action (2nd edition). Manning (2010)

[4] Krugle - Open Search, http://opensearch.krugle.org (retr. 2015)

[5] Sourcerer, http://sourcerer.ics.uci.edu/sourcerer (retr. 2012)

[6] OpenHub, http://code.openhub.net (retr. 2015)

[7] Merobase - Software Component Search Engine, http://www.merobase.com (retr. 2012)

[8] O. Hummel, C. Atkinson, and M. Schumacher. Finding Source Code on the Web for Remix and Reuse, chapter Artifact Representation Techniques for Large-Scale Software Search Engines. Springer, 2013.

[9] Susan Elliott Sim, Medha Umarji, Sukanya Ratanotayanon, and Cristina V. Lopes. 2011. How Well Do Search Engines Support Code Retrieval on the Web?. ACM Trans. Softw. Eng. Methodol. 21, 1, Article 4 (December 2011), 25 pages

[10] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. 2006. Sourcerer: a search engine for open source code supporting structure-based search. In Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications (OOPSLA '06). ACM, New York, NY, USA

[11] Siddharth Subramanian and Reid Holmes. 2013. Making sense of online code snippets. In Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13). IEEE Press, Piscataway, NJ, USA

[12] Oliver Hummel. 2008. Semantic Component Retrieval in Software Engineering. PhD Thesis, University of Mannheim

[13] Vukotic, A., Watt, N., Abedrabbo, T.: Neo4J in Action (1st edition). Manning (2014)

[14] Werner Janjic. 2014. Reuse-Based Test Recommendation in Software Engineering. PhD Thesis, University of Mannheim