Program Analysis Using Interactive and Visual Querying

Jamel Eddine Jridi, Houari Sahraoui, and Philippe Langlais
DIRO, Université de Montréal, Canada
{jridijam, sahraouh, felipe}@iro.umontreal.ca

***Abstract*—We propose an interactive querying approach for program analysis and comprehension tasks. In our approach, an analyst uses a set of basic filters (information retrieval, structural, quantitative, and user selection) to define complex queries. These queries are built following an interactive and iterative process where basic filters are selected and executed, and their results displayed, changed, and combined using predefined operators.**

***Keywords*-code querying; program analysis; visualization**

I. Introduction

Nowadays, software systems are more and more complex, which makes their maintenance very difficult. Having adequate tools for program analysis and comprehension would facilitate maintainers' work and decrease the maintenance cost. From this perspective, different generic environments for program analysis and comprehension have been proposed. They are generally based on query formulation (e.g., [1], [3], [4], [8], [9], [10], and [11]).

In these generic environments, formulating queries for many maintenance tasks is difficult for mainly two reasons. First, queries allow evaluating conditions on various entities of a program, such as classes being too complex, methods containing the keyword "auction", or methods that could be reached from a particular method. In general, these queries require that the maintainer specifies a threshold value, e.g., the level of admissible complexity of a class, TF-IDF [2] threshold for "auction", or the maximum number of calls from one method to reach another one [5]. The second reason that makes query formulation difficult lies in the fact that, in general, several basic filters have to be combined to perform an efficient search. For example, in concern-location using static analysis, structural filters are combined with information retrieval (IR) filters [5]. As these composite queries should apply to various programs, a fixed combination method generally leads to many false positives [12]. Threshold and combination problems could be alleviated if the maintainer has the opportunity to interactively and iteratively define complex queries.

In this paper, we propose an interactive environment for querying the repositories of data extracted from the source code. Our environment, named IQOP (for Interactive Querying of Object-oriented Programs), offers the possibility to use various basic filters (structural, information retrieval, quantitative and user selection). It offers also operators to combine these basic filters (e.g., set operators and iterators). Query formulation is done iteratively and the result of each step is displayed using a visualization metaphor. In our environment, querying is performed as a set of successive cycles as shown in Fig. 1. Each cycle, represented by the control-flow arrows, consists in applying a filter, inspecting the results of the filter, modifying these results, and combining them with the ones of the previous cycles (current results). For any cycle, a filter could be tried and if the results are not satisfactory, one can simply cancel it. Filter results as well as modified and current results are displayed using our visualization tool to help maintainers inspecting and modifying the results.

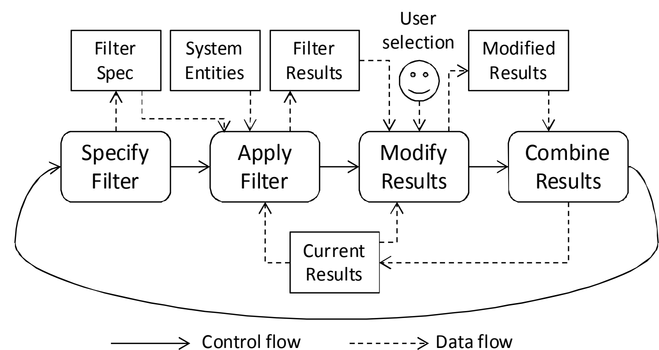

Figure 1. Overview of the querying process

II. Visualization of Filter/Query Results

It is now widely recognized that efficient visualization helps improving software data exploration. To benefit from this asset, we integrated a visualization module, adapted from VERSO [6], in order to display the querying (intermediate and final) results. As VERSO allows representing in one view thousands of classes composing a Java program, it makes it easy for analysts to inspect the elements selected by a particular filter or a set of filters. Classes are displayed as 3D boxes, placed on a plan (2D), according to the package architecture, using a Treemap layout [6]. To distinguish between classes, the values of three class metrics are mapped to three graphical attributes of the associated box: height, color, and twist. VERSO computes two dozens of metrics, but the choice of the ones to map is left to the maintainer depending on the analysis under consideration.

In a previous work, Dhambri et al. [7] extended VERSO to manually detect anti-patterns. In this context, when symptoms of an anti-pattern are evaluated, candidate classes are displayed by hiding the rest of elements. In this paper, we follow a similar approach for query visualization. Indeed, at any step of a query construction, the results of the already-

defined portion of the query are highlighted. Elements included in the results keep their original colors. By contrast, those that are not selected are still displayed but on a gray scale. Depending on the granularity level targeted by the query, this way of displaying the results applies to both classes (Fig. 2 (top)) and methods (Fig. 2 (bottom)). When the selected elements are methods, the boxes representing their classes become transparent revealing the selected methods.

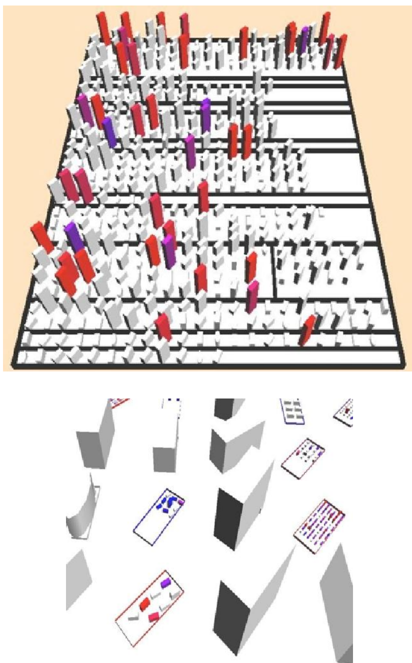

Figure 2. Visualization of filter for classes (top) and methods (bottom)

III. BASIC FILTERS

The building blocks of our querying environment are the filters. After reviewing the literature, we identified three families of filters currently in use: (1) natural-language processing, (2) structural search, and (3) quantitative filtering. As interactivity is an essential characteristic in our setting, we add a fourth family consisting in user selection.

Natural-Language Processing Filters. We implemented the three following types of filters. NAME-SEARCH filters consist in searching for a string in the names of classes, methods or both. KEYWORD filters are used to determine to which extent, elements (classes or methods) are relevant with respect to a set of keywords using IR techniques (LSI or TF-IDF). SIMILARITY filters search for code elements (classes or methods) similar to a given one. It applies IR techniques to elements' identifiers and combines the individual results using the Cosine Similarity algorithm [2].

Structural Filters. In our environment, dependencies between code elements (methods-methods, methods-classes and classes-classes) are extracted by analyzing the code statically. These dependencies include, among others, method invocation, inheritance, type reference, and inclusion. Structural filters are used to collect the elements related to a given element by a particular dependency relationship. Depending on the specified relationships, the system is viewed as a graph where nodes are the elements (classes, methods or both) and edges are the dependencies' occurrences. Hence, the filter consists simply of determining the successor of a node according to the specified dependency.

Quantitative Filters. They allow selecting, for a specific metric, a set of code elements for which the value is in a particular range. The ranges of values could be specified by particular thresholds that determine upper or lower bounds of the ranges. Alternatively, they can be defined relatively to the distribution of the values for the element set to filter. In that situation, we use the box plot technique.

User Selection. Throughout the querying process, the user can combine several types of filters. After using each filter, she can inspect the results returned and decides to which extent the filter specification is relevant. For example, consider the program ArtOfIllusion, a 3D modeling and rendering studio (http://www.artofillusion.org/). If a maintainer is looking for classes participating in the image encoding functionality (formats such as JPEG and GIF), she can start by applying a name search filter with string "encoder". The result of this filter will be a set of classes such as JPEGEncoder and BMPEncoder. Structural filters are then used to add classes that are linked to these classes. When inspecting the classes returned by the name search filter, the maintainer could make three decisions: (1) the majority of the returned classes are not relevant, which leads to cancel the filter or refine the search criteria, (2) all the returned classes are considered as relevant, which leads to continue the search with structural filters, and (3) the majority of classes are relevant, but some are missing (no class for GIF encoding) or are not relevant (a class WidgetEncoder was found). In this case, the maintainer could select a class, she knows it encodes GIF images, and adds it manually to the result. She can also remove WidgetEncoder from the result. Once these alterations are done, she proceeds with the structural filters.

IV. COMBINING FILTERS

When performing analysis and comprehension tasks, complex queries are expressed by combining basic filters. Basic filters allow exploring search criteria of different natures: lexical, structural and quantitative. Their combination helps increasing the precision of the analysis. Maintainers could combine filters using classical set operators and/or iterators (interleaving combination).

For set-operator combination, we distinguish between unary and binary operators. The only unary operator we use is COMPLEMENT. This operator is used to express negative search criteria. It returns all elements that are not included in the result of a filter. For example, if we want to exclude from our analysis exception classes, we could apply the name-search filter with string "exception", and then apply the COMPLEMENT operator to consider all the classes but those implementing exceptions.

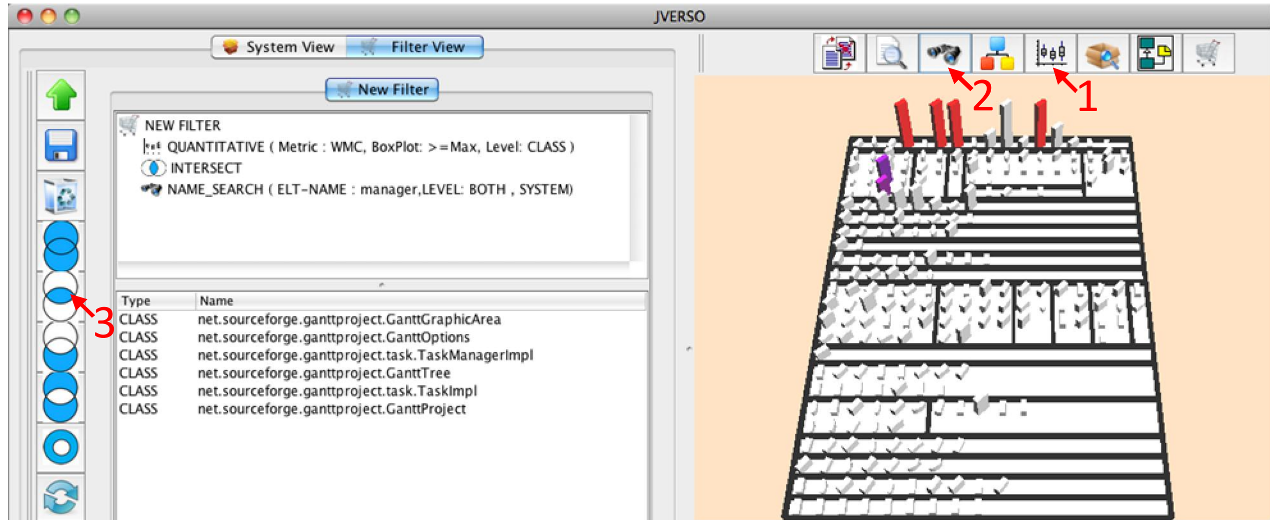

Figure 3. An example of combining two filters.

Four binary operators are provided to combine filter results. These operators are UNION, INTERSECTION, DIFFERENCE, and SYMMETRIC DIFFERENCE. For instance, if we are searching for classes with a very high complexity and that have the string "manager" in their names, we could apply the quantitative filter using the metric WMC and range higher-than-upper-tail (item 1 in Fig. 3) and then, the filter name-search with the string "manager" (item 2). Finally, both results are combined using the INTERSECT operator (item 3).

Interleaving combination allows applying a filter to each element of the result set returned by another filter. To this end, we use iterators. Our environment provides two iterators, the basic and the conditional ones. The basic iterator allows to apply the structural filter recursively. Starting from an entry point, it selects the elements related to this entry point by a specified relation. Then, it considers each selected element as an entry point and applies to it the same filter, and so on and so forth. All the elements selected at all the levels are added to the final result. For example, to collect all the descendents of a class, one could simply apply an iterator on this class with, as a structural filter, the relationship subclass.

A conditional Iterator is similar to the basic one with the exception that at each level of the considered graph, the structural filter is recursively applied only to elements that are selected by another filter. This second filter could be, for example, a keyword, a similarity, a name-search, or a quantitative filter. Like the basic one, the conditional iterator could be applied to all the levels without interuption. It can be applied as well level by level, allowing the maintainer to inspect and to alter the results at each level. To illustrate the use of the conditional iterator, let us consider the situation where a maintainer is searching for inheritance paths, in the collection hierarchy, where all the classes have "add" methods. Like for the example of the basic iterator, she can iterate on the subclasses starting from the hierarchy root. However, using the conditional filter with name-search on "add", at each level, only classes that have "add" methods will see their children explored.

V. Application Examples

To illustrate how our environment could be used in maintenance, we present two examples of maintenance tasks: (1) detection of design defects and (2) location of concern implementation in the code.

A. Design Defect Detection

In DECOR [12], Moha et al. describe the symptoms of each defect type using an abstract rule language. For example, the defect blob is described in DECOR as a large controller class with low cohesion related to several data classes. Four metrics are used: LCOM for lack of cohesion in the blob candidate, NMD+NAD (numbers of declared methods and attributes) for its size, and NACC (number of accessors) as an indicator for data classes. Each of these metrics is associated to a threshold value that determines if a quantitative symptom is found. Lexical information is extracted from the names of classes and methods to check the presence of terms such as "manager", "controller", and "process". Finally, dependencies between classes are used to verify if a blob candidate is associated with data classes. Here again the number of data classes should exceed a certain threshold. Conditions about symptoms are combined using conjunction and disjunction operators.

We evaluated the symptoms described by Moha et al. [12] to detect blobs using IQOP on the open-source Java project Gantt v1.10.2. The evaluation of metric symptoms was done through quantitative filters considering classes having extreme values with respect to the boxplot distribution (values higher than the upper tail). To assess if a class plays the role of a controller, we use the name search filters on both the class and its methods with the list of terms used by DECOR. To combine the above-mentioned symptoms, we use INTERSECT and UNION operators for

respectively the conjunctions and disjunctions. Finally, the potential links between a blob candidate and data classes are checked using the conditional iterator on the dependency relationship. The condition of the iterator is defined by a quantitative filter on the number of accessors (NACC) with as a range, greater than the upper tail. The iterator is applied to the first level of the dependency graph because only immediate candidate-related classes are considered.

This straightforward query formulation with IQOP gives the same results as the original implementation of DECOR. However, when using the interactivity and result inspection, the analyst could improve the detection results. For example, when searching for classes playing the role of a controller with a name search of the term "controller", only two classes are found, but neither are blobs. When changing the filter to keyword search with LSI (with the same term), nine classes are found and three of them are actual blobs. The other six classes are eliminated when combining with the quantitative filter on complexity.

B. Concern Location

In DORA [5], Hill et al. combine both program structure and lexical information techniques to help programmers tracing requirements to code. This is done in four steps: (1) extract the list of direct neighbors in the call graph of a starting method, (2) calculate the lexical relevance score of neighbors with respect to a keyword search, (3) remove methods with a score below a threshold, and (4) apply steps 1-to-4 recursively to the remaining methods. For example, consider the correction of a portion of code that triggered a bug when adding an auction in the program JBidWatcher v1.0pre6. As adding an auction is an action, the entry point for exploring the call graph is the method DoAction(). For the keyword search, the terms to use are "add" and "auction". The two methods that should be found and corrected are DoAdd() and DoPasteFromClipboard().

We implemented the concern-location technique according to the choices made by DORA's authors. We combined structural exploration and keyword search using the conditional iterator. This iterator explores the call graph using the call_method relationship. The condition on the exploration is defined by the keyword-search filter with TF-IDF. With this implementation and for a particular threshold value, we found the two expected methods *DoAdd()* and *DoPasteFromClipboard()* of the above-mentioned example. However, slight variations in the keywords and/or score threshold values could lead to different results.

Interactive querying could help improving the results of concern location. First, the call graph could be explored level by level, and the analyst could stop the exploration at any moment if she judges that relevant methods are already found. Second, different keyword search variation could be explored. For example, compared to TF-IDF, LSI could find methods that do not have the exact searched terms, but terms semantically close to those. Increasing or decreasing the score threshold could also help finding more positive methods or removing false-positive ones. The third possible improvement could be achieved through the visual inspection of the results at any step of the exploration.

VI. Conclusion

In this paper, we have presented a generic environment for software analysis and comprehension using querying mechanisms. In contrast with previous contributions, our environment, in addition to be generic to a large spectrum of comprehension tasks, involves the analyst in the exploration process. This is done by an interactive visualization environment.

The most important of our contributions, is to support the analyst in writing and refining queries, as well as in inspecting and altering the results of those queries. Indeed, in many situations, the definition of fixed automated processes is not realistic considering the variety of software systems and the lack of knowledge in many comprehension tasks. Introducing interactivity with efficient visualization metaphors helps improving the precision of comprehension tasks for large software systems, while minimizing the cost of human interventions.

References

[1] B. de Alwis and G. C. Murphy, "Answering conceptual queries with Ferret", Int. Conference on Software Engineering, pp. 21–30, 2008.

[2] C. D. Manning, P. Raghavan and H. Schütze, "Introduction to Information Retrieval", Cambridge Univ. Press. 2008.

[3] D. Janzen and K. de Volder, "Navigating and querying code without getting lost", 2nd Int. Conference on Aspect-Oriented Software Development, pp. 178–187, 2003.

[4] E. Hajiyev, M. Verbaere, and O. de Moor, "CodeQuest: scalable source code queries with Datalog", Eur. Conference on Object-Oriented Programming, pp. 2–27, 2006.

[5] E. Hill, and K. Vijay-Shanker, "Exploring the neighborhood with DORA to expedite software maintenance", Int. Conference on Automated Software Engineering, pp. 14–23, 2007.

[6] G. Langelier, H. Sahraoui, and P. Poulin, "Visualization–based Analysis of Quality for Large–Scale Software", Int. Conference on Automated Software Engineering, pp. 214–223, 2005.

[7] K. Dhambri, H. Sahraoui, and P. Poulin. "Visual Detection of Design Anomalies". Eur. Conference on Software Maintenance and Reengineering, pp. 279-283, 2008.

[8] M. Consens, A. Mendelzon, and A. Ryman, "Visualizing and Querying Software Structures". 14th Int. Conference on Software Engineering, pp. 138–156, 1992.

[9] M. Marin, A. Van Deursen, and L. Moonen, "SoQueT: Query-based documentation of crosscutting concerns", 29th Int. Conference on Software Engineering, pp. 758–761, 2007.

[10] M. Verbaere, E. Hajiyev, O De Moor, "Improve software quality with SemmleCode: an eclipse plugin for semantic code search", Companion to OOPSLA, pp. 880–881, 2007.

[11] M. Würsch, G. Ghezzi, G. Reif, and H. Gall, "*Supporting Developers with Natural Language Queries",* Int. Conference on Software Engineering, pp. 165–174, 2010.

[12] N. Moha, Y.-G. Guéhéneuc, L. Duchien, A.-F. Le Meur. "DECOR: A Method for the Specification and Detection of Code and Design Smells", IEEE Trans. on Software Engineering, vol. 36, no. 1, pp. 20–36, 2010.