Enhancing Static Source Code Search With Dynamic Data

Reid Holmes and David Notkin
Computer Science & Engineering
University of Washington
Seattle, WA 98195-2350 USA
rtholmes,notkin@cs.washington.edu

ABSTRACT

Developers frequently try to locate references to particular program elements within their systems; however, these queries often return an overwhelming number of results. The result sets for these queries tend to be large because integrated development environments locate matches using static search approaches; however, the developer may be more interested in which references *actually* happened for a particular execution, instead of which references *could* happen in a hypothetical execution. We posit that dynamic search approaches can complement customary static search approaches in the same ways dynamic analysis complements static analysis. Specifically, in this paper, we hypothesize that filtering static reference queries with dynamic trace data can reduce the number of results a developer must consider when performing a query, helping them to focus on a subset of the static query results. To test our hypothesis, we filtered the results of the Eclipse *find references* query with dynamic trace data for three different projects; our preliminary evidence demonstrates that dynamic trace data can be used to effectively filter the result sets of static source code queries.

1. INTRODUCTION

While being able to statically query for all references to a specific method is valuable, the number of results a developer must consider can be overwhelming; in this paper we provide preliminary evidence demonstrating that dynamic filtering can significantly reduce the number of results for these queries. While a sound static approach can identify all program locations where a method *could* be called during any given execution, a dynamic approach instead fully captures all program points that a method *was* called from in a specific execution. The elements surfaced by a dynamic approach are particularly suitable for a developer investigating a specific subset of a system (e.g., a specific test or specific part of a user's trace).

Finding all of the references to a specific program element within an Integrated Development Environment (IDE) is a frequent activity; a previous study has found that more than 80% of Eclipse developers perform this type of activity [2]. This type of query corresponds well to a conceptual question a developer may ask while modifying or investigating a software system, namely, 'where is this method called?' [5].

At least two previous approaches have integrated dynamic trace data with static search tools. While both approaches enable developers to search for method references using dynamic data, these were minor features in larger systems; the empirical benefits of applying this approach were not explicitly explored.

Ferret [1] consists of a general model for integrating different sources of query data; one of the 27 different queries Ferret supported enabled developers to query for references in a dynamic call graph. In a diary study, one of their participants specifically noted the value of knowing a result had been executed: "[seeing] what was *actually* called was useful because it eliminated spurious calls made from [other contexts]." Alternatively, Hermion [3] uses runtime trace data to answer source code queries for Smalltalk; this approach is especially useful for dynamic languages as static search approaches are not as effective for these languages. The Senseo [4] Eclipse plug-in adds additional dynamic data to the Eclipse IDE, particularly in the source code editor, but does not augment search results with dynamic data.

This paper contributes preliminary evidence demonstrating that dynamic filtering can effectively reduce the number of results for static queries which can help developers focus on specific query results. Section 2 provides a short scenario describing how a static query can quickly return results that are overwhelming. A brief overview of our approach is outlined in Section 3. Section 4 provides preliminary evidence for our approach while Section 5 provides discussion and future work. Section 6 concludes.

2. MOTIVATION

While IDE support for locating references to particular types, fields, and methods is extremely valuable, oftentimes the results of these queries can be overwhelming. Figure 1 shows the partial results of a find references request, made by a developer named Zoe, for a method in their system; in this example more than 1000 results are returned spanning 84 packages.

In practice, when Zoe queried for the references to `getElementName()`, she was working on a specific bug; although 1,076 static results were returned, some guidance to help her decide which of these to investigate first would be help-

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

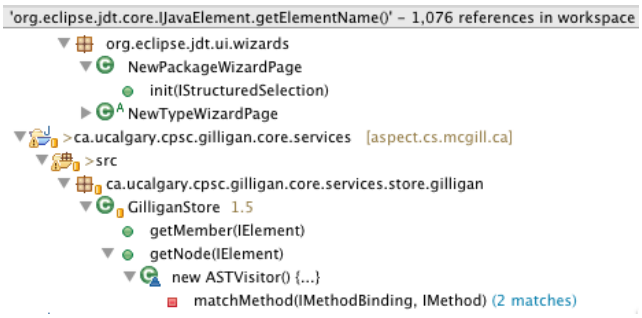

Figure 1: Overwhelming query result.

ful. Applying a dynamic filter based on the last execution of the system narrows the number of results to 16 (a 98% reduction), which is more manageable. These results also make more sense to her as they are in the context of a specific execution with whose inputs she was familiar.

3. APPROACH

We built our prototype for the Eclipse IDE, collecting static search results in the same manner the IDE itself does for finding references to method calls. While identifying type and field references is also possible, the current version of our tool only records and analyzes method call relationships.

Instead of performing the queries manually and scraping the results we instead used the `SearchEngine` class using the `SearchUtils.GENERICS_AGNOSTIC_MATCH_RULE` match rule and collected the results programmatically.

The dynamic call graph was collected using a custom AspectJ aspect that maintained a call stack as the test harnesses executed. While the aspect would work with any execution, we only used it with test harnesses for this application.

Static results can be extracted directly from the `SearchEngine` results; the dynamically filtered results comprise those results that intersect both the static results and the dynamic call graph. While in some cases elements may be present in the dynamic call graph and not the static call graph, we do not currently consider these elements. Some of these elements arise due to reflection, others due to annotations (e.g., `@Test` indicates a JUnit test will be executed whereas the edge from JUnit to the test will not appear statically). While our system can match static search results with elements from the dynamic call graph with fairly high precision it is not perfect and some matches may be missed in this process.

4. PRELIMINARY EVIDENCE

We were interested in gathering evidence for two main questions: (1) Can dynamic filtering effectively reduce the number of results for queries that might otherwise be overwhelming (Section 4.1)?; and (2) In aggregate, how much can dynamic filtering reduce the number of results for a source code query (Sections 4.2 and 4.3)?

We tested our prototype on three open-source systems that contained JUnit test harnesses. While the test harnesses were not required, the test harnesses ensured we could test our filtering on consistent execution data. Our test systems were JodaTime, the Google Visualization Library, and a reference implementation for RFC-2445[1]. These projects comprise 76 KLOC, 17 KLOC, and 7 KLOC and contain 2,525, 365, and 171 tests respectively.

To evaluate our approach we collected the static search results and dynamic call graph for each of the systems we investigated. To collect the static search results, we programmatically invoked the Eclipse find references command on every method in each system and recorded the results; we configured the workspace to ensure that only the project being searched and its required libraries were present to reduce the chances of incorrect results being returned. The dynamic call graph was collected by executing the complete test harness for each system and recording the execution's dynamic call graph.

4.1 Addressing Overwhelming Queries

Developers will investigate a limited number of query results before giving up or researching [6]. In light of this, while decreasing the number of query results from 4 to 2 is a 50% reduction, filtering a different query from 15 to 10 can have more value if a relevant result was in the subset they would not have otherwise investigated. As such, from a developer's perspective, we expect reducing query result sizes below their threshold of patience may be more important than the overall reduction rate.

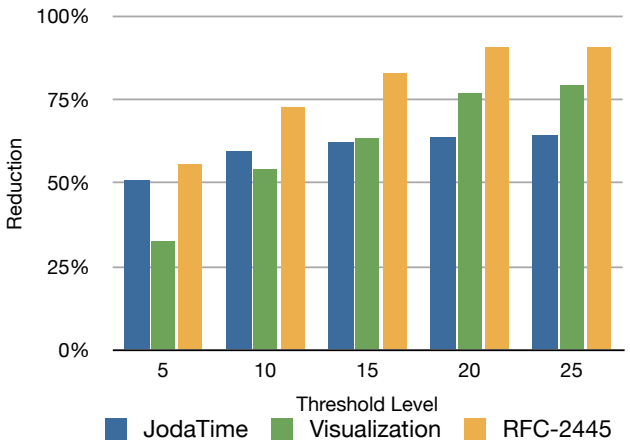

Figure 2: Percentage of over-threshold queries reduced below threshold by dynamic filtering.

Figure 2 reflects how many queries, over various thresholds, are reduced below the threshold level by dynamic filtering. For example, for a threshold of 10 (e.g., for all queries where the number of static results is greater than the threshold level), applying dynamic filtering reduces the number of results to below the threshold for more than 50% of these over-threshold queries. As the queries that return a large number of results most require filtering, Figure 2 shows that dynamic filtering can effectively help reduce the number of results for a large proportion of these queries.

[1]http://code.google.com/p/google-visualization-java/, http://joda-time.sf.net, http://code.google.com/p/google-rfc-2445/.

4.2 Aggregate Results

We gathered aggregate results to see if dynamic filtering data decreased the number of results a developer would have to consider for any method in a system. Table 1 shows the results for each of our evaluated systems. The second column represents the total number of results for the static search queries for every method in the system that executes at runtime. The third column represents the total number of results for this same set of methods when the static results are filtered by the dynamic results; the fourth column contains a percentage representation of this information reduction.

Project	Executed Methods	Static Results	Filtered Results	% Reduct.
JodaTime	5,047	94,230	17,616	84%
Visualiz.	1,093	18,167	2,761	87%
RFC 2445	416	12,791	732	95%

Table 1: Aggregate query result reduction.

While these findings seem positive, examining the results manually we noticed that while the dynamic filtering made huge improvements for some queries, for many others it did not seem to make any difference at all. To investigate this, we further broke down the results by method to determine how much the search results were reduced for each method. We split the reduction into four buckets: no improvement, $0 < reduction \leq 33\%$, $33\% < reduction \leq 66\%$, and $66\% < reduction \leq 100\%$. These results are shown in Figure 3.

Figure 3 shows that while a significant proportion of methods results do not receive any benefit from dynamic filtering, some methods do benefit strongly from the dynamic filtering approach. In combination with Table 1, this indicates that those methods that do benefit experience very large reductions while other methods do not benefit at all.

Two main factors account for the difference between Table 1 and Figure 3. First, dynamic filtering can make a huge difference for some methods. In some instances, methods that have a common signature or a part of a popular interface (e.g., `equals(Object), get(Object), and add(Object)`) are often matched by the conservative static analysis, even if they are not called in practice. Secondly, many methods have very specific names that don't match any results in the static search. These methods exist by the hundreds in these systems: most of the test suite methods have unique names like `testQueryDoesntRuinDataSourcePatterns()`; these methods contribute to the high proportion of 'no reduction' methods. Inspecting the 'no reduction' partition more closely shows that the Eclipse find references command returns no static results for 49.9% of the methods, precluding any further reductions using dynamic filtering. While the dynamic call graph may contain actual results for the methods that have no static results, our evaluation has concentrated only on refining static results, not adding new results that are not statically observable.

4.3 Partial System Results

Developers performing a task typically only work on a subset of the system, as such we were interested to see if the results of applying dynamic filtering were different when applied to a constrained portion of the system. To do this,

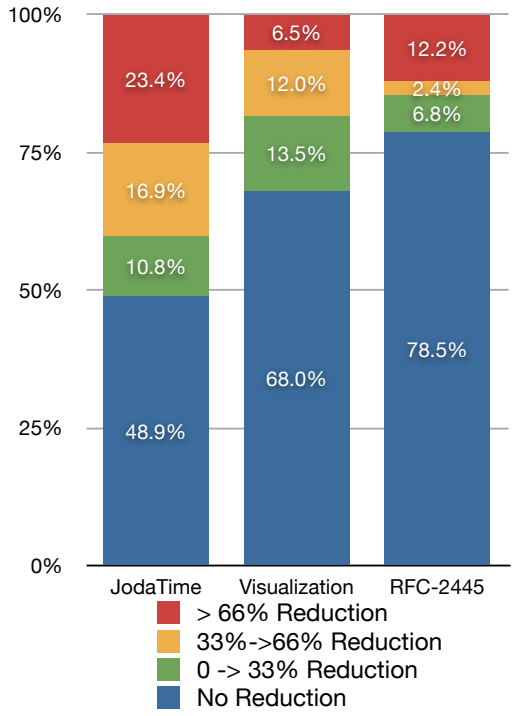

Figure 3: Amount of query result reduction per queried method;

we decided to select three unit test classes at random from each project, applying dynamic filtering only to the subset of methods that were executed by the tests. Figure 4 shows the clustering for these nine cases.

The partial-system results seem more promising than the whole-system results. For each task the majority of methods received some amount of reduction when the dynamic filtering was applied and the proportion of methods that received a large amount of filtering increased. While the test methods (that do not benefit from filtering) are still executed and included, they only comprised a subset of the executed methods. These results also highlight another potential use for applying dynamic data in an IDE: a developer could specify that they are working on a specific test class and the IDE could then provide them with a 'working set' that is pre-populated with the methods that are executed by that test class to help them identify the subset of the system that may be relevant to their task.

5. DISCUSSION

5.1 Validity

While our preliminary evaluation showed that dynamic filtering could reduce result sizes, this preliminary evidence should be more rigorously augmented. The largest threat to the validity of our preliminary investigation is that we have not validated whether relevant results are elided by dynamic filtering; this remains for future investigation. The three systems we evaluated with were from different domains, but to enhance the external validity of our filtering result, testing this approach on additional, larger, systems would be beneficial.

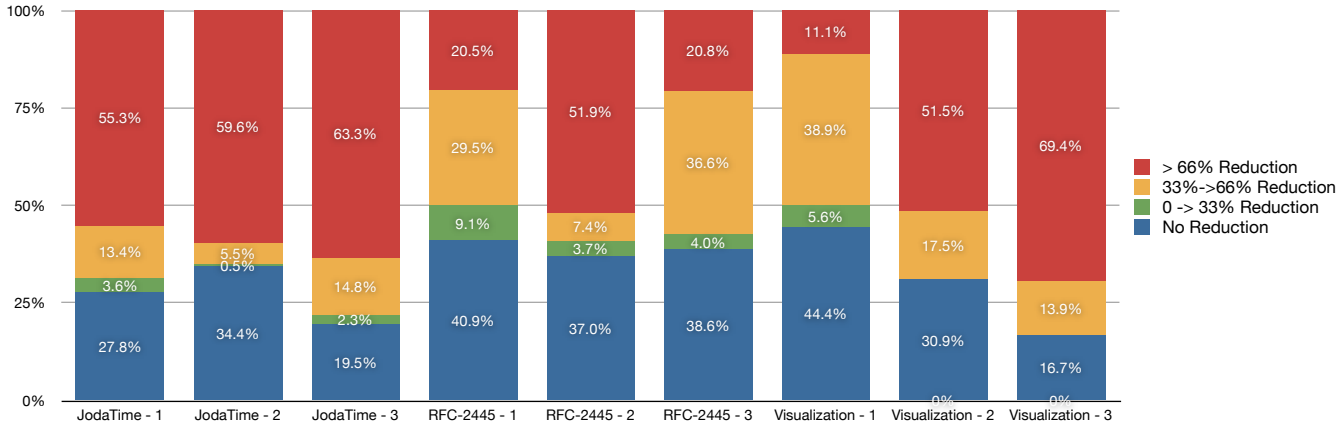

Figure 4: Amount of query reduction for methods executed by randomly-selected test classes.

5.2 Presenting Filtered Results

Dynamic filtering may not be the right default behaviour for finding references in the IDE; however, results that match the dynamic filter could be decorated in a normal result list (for instance in bold). A toggle to enable the dynamic filter would be a lightweight addition to the default search view. An interesting alternative would enable the developer to choose the execution they want to filter with (e.g., the last run or the run from two weeks ago) and enable them to contrast the dynamic filter results between these time periods (e.g., did the same methods reference this method last week as are doing so now?).

5.3 Future Work

Our prototype tool does not provide any UI for the user; before any evaluations involving users are conducted, a UI integrated with the Eclipse search functionality must be created. Extending dynamic filtering to other queries (e.g., field and type references) is also a logical next step. To better support this, and to improve the performance of generating the dynamic data, we are currently working on a new dynamic tracing approach that is more efficient and better supports multiple threads.

6. CONCLUSION

In this paper we have demonstrated the utility of filtering static search results in an Integrated Development Environment using dynamic data. Dynamic filtering enables the developer to focus their efforts on those results that are most likely to be relevant to them. This approach provided significant reductions in the number of search results a developer might need to consider. Our preliminary evidence suggests that for many queries the number of results can be filtered by greater than two thirds and that dynamic filtering can reduce over half of queries returning more than ten results to under that threshold. On the basis of this evidence, further investigation into applying dynamic filtering to static search results in IDEs is warranted.

7. REFERENCES

[1] B. de Alwis and G. C. Murphy. Answering conceptual queries with Ferret. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 21–30, 2008.

[2] G. C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the eclipse ide? *IEEE Software*, 23(4):76–83, 2006.

[3] D. Rothlisberger, O. Greevy, and O. Nierstrasz. Exploiting runtime information in the ide. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 63–72, June 2008.

[4] D. Rothlisberger, M. Harry, A. Villazon, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret. Augmenting static source views in ides with dynamic metrics. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 253–262, 2009.

[5] J. Sillito, G. C. Murphy, and K. D. Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, pages 23–34, 2006.

[6] J. Starke, C. Luce, and J. Sillito. Searching and skimming: An exploratory study. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 157–166, 2009.