# Introducing Natural Language Program Analysis

Lori Pollock, K. Vijay-Shanker, David Shepherd,
Emily Hill, Zachary P. Fry, Kishen Maloor
Computer and Information Sciences
University of Delaware
Newark, Delaware 19716
{pollock, vijay, shepherd, hill, fryz, maloor}@cis.udel.edu

## 1. INTRODUCTION AND MOTIVATION

Throughout the life cycle of an application, between 60-90% of resources are devoted to modifying the application to meet new requirements and to fix faults [1]. Building effective software tools is important to reduce these high maintenance costs. In our research, we have observed strong indicators that there are many natural language clues in program literals, identifiers, and comments that could be leveraged to increase the effectiveness of many software tools.

For the past two years, our research group has been investigating how to best extract and utilize natural language clues from code. We call this kind of analysis, Natural Language Program Analysis (NLPA), since it combines natural language processing techniques with program analysis to extract natural language information from the identifiers, literals, and comments of a program. Using NLPA, we have developed techniques and integrated tools that assist in performing software maintenance tasks, including program understanding, navigation, debugging, and aspect mining.

Thus far, we have focused on NLPA tools that identify scattered code segments that are somehow related: whether it be to search through code to understand a particular concern implementation, to mine aspects, or to isolate the location of a bug. Our existing NLPA tools combine program structure information such as calling relationships and code clone analysis with the natural language of comments, identifiers, and maintenance requests. Although we have only begun to explore the potential of NLPA, our various experimental results motivate further investigation of NLPA for software tools.

We believe that NLPA can be used to (a) increase the accuracy of software search tools by providing a natural language description of program artifacts to search, (b) increase the ability of program navigation tools to recommend related procedures by providing natural language clues, and (c) increase the accuracy of other program analyses by providing access to natural language information.

## 2. PRESENTATION OVERVIEW

This group presentation will begin with examples to demonstrate the valuable natural language information that can be extracted from programs. We will then present an overview of our program representation for NLPA and the extraction process to build this representation. A student will present our lessons learned in natural language clue extraction and his current work on improving the extraction process. To show how this analysis can be used to improve a variety of software tools, students will then describe the set of tools we have built using NLPA, including Find-Concept [3] for search and navigation, Timna2 [4] which is a machine learning-based aspect miner in which we have recently integrated NLPA with traditional program analysis features, and CallGraph-Explorer, which combines NLPA-based heuristics to identify a relevant call graph window for program comprehension and maintenance. The presentation will conclude with some posed questions for discussion. We now provide an overview of each short presentation.

## 3. APPLYING NLPA

### 3.1 Concern Location

The research community agrees that object-oriented programming, which can be viewed as noun-oriented, causes concerns to become scattered [2, 6], and we argue that many of these scattered concerns are *action-oriented* because of the natural tension between objects and actions [5]. Thus, we have focused on NLP analysis that captures the relations between actions (verbs) and the objects (nouns) that these actions act upon, i.e., direct objects in English. We have been using this NLP information to connect related, scattered code segments, which often have long, obscure chains of structural links between them.

In order to leverage this information about programs, we process source code to extract verb-DO pairs. We have designed a novel program model, the **action-oriented identifier graph (AOIG)**, that represents the occurrences of verbs and direct objects (DOs) in a program, as implied by the usage of user-defined identifiers [5].

The particularities of source code structure can frustrate a search engine's ability to find relevant code segments. We use NLPA to deduce each method's purpose and create an easy-to-search source code representation, the AOIG. For instance, a direct search over source code, such as the lexical search "add auction", will not find all relevent entries, such as method add_entry (which actually adds an auction to a list). We use NLPA to determine that the method add_entry actually "adds an auction" by analyzing its parameter types (type AuctionInfo).

We have created a prototype code search tool, **Find-Concept**, that searches the AOIG instead of unprocessed source code, often leading to more effective searches [3]. Find-Concept also leverages the AOIG's structure and NLP techniques to assist the user in expanding her initial query into a more effective query (e.g., by using synonyms). In an experimental evaluation, Find-Concept performed more consistently and more effectively than a state-of-the-art com-

petitor (a plugin to Google Desktop).

Our work on Find-Concept exposed some limitations of our initial AOIG Builder [5] for extracting verbs and their direct objects from source code. In our initial work, we found that common method naming conventions follow a relatively small number of patterns, and so we created a set of rules to identify instances of each pattern. Generally, we extracted incorrect pairs for two reasons: either the statistical part-of-speech tagger component of the AOIGBuilder failed due to programming specific terms or, more commonly, no rule existed for a given naming convention.

We have been adding new rules to account for previously unknown naming conventions and also manually supplemented the part-of-speech tagger to correctly tag words specific to programming. For example, our initial approach would return the verb-DO pair <run, unknown> for the method run() in the class IReportCompiler. However, we believe that the pair <compile, ireport> is more appropriate because the method is "running the IReportCompiler" which reduces to "compiling the ireport". Through identifying commonly occurring patterns such as this we hope to generate a more correct and comprehensive set of verb-direct object pairs for a given segment of source code.

## 3.2 Aspect Mining

To realize the benefits of AOP, developers must refactor active and legacy code bases into an AOP language. When refactoring, developers first need to identify refactoring candidates, a process called aspect mining. Humans mine using a variety of program-analysis-based features, such as call graph degree. However, existing automatic aspect mining approaches only use a single program-analysis-based features. Thus, each approach finds only a specific subset of refactoring candidates and is unlikely to find candidates which humans find by combining characteristics. We have created a framework, Timna, that uses machine learning to combine program analysis characteristics and identify refactoring candidates. Previously, we evaluated Timna and found that it was an effective framework for combining aspect mining analyses [4].

Most previous work in automatic aspect mining, including Timna, leverage program analyses exclusively. To evaluate Timna's effectiveness, we annotated every refactoring candidate in a substantial Java program. We began identifying candidates using only program analyses clues, yet we found that often natural language clues reinforced the program analyses clues. For instance, two methods that are often called in the same method body are often refactoring candidates. These methods also usually had names which were opposites of each other, such as "open" and "close". **Timna-2** uses these program analyses and natural language clues together to more effectively identify refactoring candidates. In our initial evaluation, Timna-2 (i.e., Timna with NLPA features added) reduces the misclassification rate by 38%, as compared to Timna.

## 3.3 Program Exploration

By visualizing calling relationships of a program, a software maintainer can quickly identify and comprehend large portions of code relevant to a maintenance task. Unfortunately for today's software sizes and complexity, call graphs are prohibitively large and unwieldy for use by the software engineer. We have been examining how natural language program analysis, combined with program structure and context information, can be leveraged to focus a software engineer on a *relevant window* of a program's call graph.

We have developing a prototype Eclipse plug-in, **CallGraph-Explorer**, that creates a relevant call graph window and displays the window to the software engineer. Starting with a seed method and a natural language description of the bug to be fixed or feature to be added, our CallGraph-Explorer tool limits how many calling edges are included in the relevant call graph window in any direction (from callers or to callees) by comparing how relevant the terms used in the method and comments are to the natural language description of the maintenance task. Initial evaluation with CallGraph-Explorer suggests that even just considering the relevance of method names can significantly limit the call graph window to a viewable size, while including all relevant methods to the maintenance task at hand.

# 4. CONCLUSIONS

We have conducted several experimental studies to evaluate our initial use of NLPA. Our results are promising, motivating further investigation of NLPA for software tools and exploring other applications of NLPA. Specifically, we pose the following discussion points for the PASTE community:

- What open problems faced by software tool developers can be solved or mitigated by NLPA?
- How might NLPA be applied to current research projects to advance the state of the art?
- Under what circumstances is NLPA not useful?

# 5. REFERENCES


[1] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[2] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP*, 2001.

[3] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to find and understand action-o riented concerns. In *Int. Conf. on Aspect-oriented Software Development*, 2007.

[4] D. Shepherd, J. Palm, L. Pollock, and M. Chu-Carroll. Timna: A framework for combining aspect mining analyses. In *International Conference on Automated Software Engineering*, 2005.

[5] D. Shepherd, L. Pollock, and K. Vijay-Shanker. Towards supporting on-demand virtual remodularization using program graphs. In *Proc. of Int. Conf. on Aspect-oriented software development*, 2006.

[6] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *ICSE*, 1999.