Senior Design

Sociotechnical Solution to Large-Scale Formal Specification Mining

Final Report

Group May1620

Alex Dana, Cody Hanika
Advisor/Client: Professor Rajan

Table of Contents

Figures

Overview

Java specifications are precise definitions of methods of Java libraries. They have the potential to benefit all programmers. However, formal specifications are hard to come by and often times difficult to produce. Formal specifications involve describing methods in mathematically precise language and verifying preconditions and postconditions. Hence, the purpose of this project is to create a socio-technical solution to this problem. The problem will be solved by using parsing informal specifications from javadoc which we will then analyze with a natural language parser to produce valid formal specifications.

Problem Statement/Background

Formal specifications provide a generalized, high-level, and easily extendable approach to verification. Through the use of formal specifications, it is easier to write correct code and to avoid the use of an API in a manner that was not intended. Currently, popular Java libraries either lack publicly available formal specifications or it is difficult to obtain these specifications. Previous attempts at creating specifications at a large scale has failed due to the difficulty of generating correct formal specifications from code without human intervention.

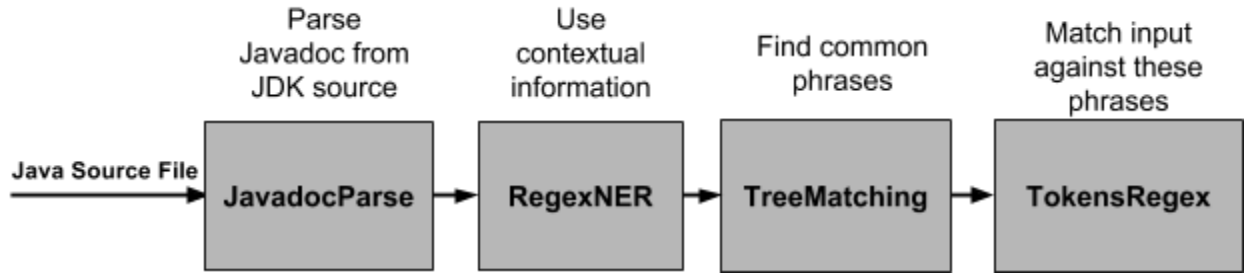
System Level Design

- The application must be able to utilize the Stanford Natural Language Processor to analyze informal specifications in Javadoc
- Rules will be provided via regular expression files to form JML (Java Modelling Language) specifications from the analyzed informal specifications
- The system must be able to run this process over the entire JDK if desired
- The process must run in a reasonable amount of time (undefined time but preferably under 5 minutes)

There are 4 main parts of the application.

1. **JavadocParse** - This is where we extract javadoc from sourcecode
2. **RegexNER** - Give the natural language processor some context of what is important for our analysis
3. **TreeMatching** - Determine the most common informal specification structures
4. **TokensRegex** - Turn the common informal specification structures into JML

Figure 1: This is a basic diagram showing the communication flow and design for the project



Detailed Description

Major Components

JavadocParse: This class will be responsible for taking source code and parsing out the javadoc. The process can be simplified as:

1) Verify directory of JDK source code
2) Create Abstract Syntax Tree
3) Create and use DocumentationVisitor

RegexNER: This component is more blackbox compared to the others due to the fact that we heavily utilize the Stanford Natural Language Processor to process our informal specifications obtained via the JavadocParser. The analysis creates parse trees that are assigned parts of speech and named entity recognizer (NER) tags which we can use for later processing.

TreeMatching: This class is responsible for analyzing the JDK to find common subtrees which represent patterns in informal specifications. The common informal specifications can then be tackled first to create JML.

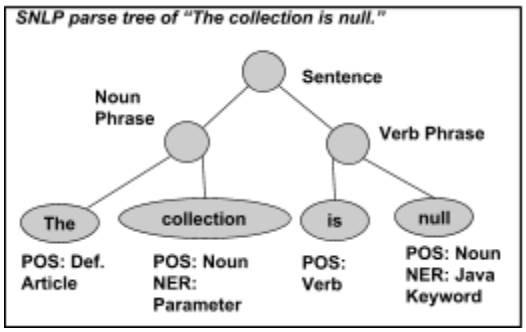


Figure 2: The tree structure NLPProcess produces for each informal specification

TokensRegex: This class will be responsible for interpreting our regex rule files in order to process and create valid JML specifications from informal specifications. This is done as follows:

1. Informal specification as well as parameter types/names are provided
2. Rule files are loaded in
3. Informal specification is analyzed to see if it conforms to any rules
4. If so, JML is generated based on rules and parameters

Risks

There were many risks for us regarding this project. Our approach outlined here was only taken after a previous approach was unsuccessful (see Appendix II) meaning that we had limited time to produce results. Our lack of knowledge with natural language parsers made for a steep learning curve. These points paired with the fact that we were not entirely sure that the javadoc would contain enough information for us to create specifications from made for a high risk project.

Testing/Results

Testing was done via manual statistical analysis of results. We performed analysis on as many types of common informal specifications as possible (this involved NullPointerExceptions along with other boolean type expressions). Overall our results were very promising and show a definite basis for the continuation of our project in the future. The following table shows the full results for the core java packages:

Java package	**Throws Documentation Match %**
awt	22.9%
io	10.7%
lang	25.2%
math	19.2%
net	11.3%
nio	11.7%
security	21.1%
time	1.6%
util	49.8%

The results are quite varied based on the kind of code the documentation was describing. Our analysis focused on mathematical expressions and ensuring we covered all NullPointerExceptions. Some of these packages, such as security, use these kinds of things for specifications quite a bit. Others, such as io, have a lot of documentation that is less mathematical and describes system properties that might cause an exception, such as a file not existing. These things are much harder to parse out of the documentation successfully. These initial results show that focusing on certain types of exceptions can be a successful way to find specifications.

Current Progress/Future Plans

At the moment all components are in working order. The creation of rule files is something that is done manually (for now) and therefore is a slower process. Because of this we currently only support JML specifications regarding mathematical exceptions as well as null pointers. However, as our results show above, this was highly successful. The project could easily be continued by another group who would need to add rules to handle new informal specification structures. The automatic generation of rule files is also something that can be considered. With more time and knowledge we believe that automatic (or at least partial) generation of rule files is a feasible direction to take this project in.

Appendix I: System Setup

Our project is setup to help get things running as fast as possible with as little time spent finding dependencies as possible. However there are some steps that will need to be taken in order to complete the setup. By following these instructions the setup should be fairly fast and painless.

Downloads:

Our project repository is located at https://github.com/may1620/spec/tree/finalProduct. Our finalProduct branch should have the most stable release while other branches can be used to try out unfinished/in progress features. Once the download is complete the source files for a recent JDK will be needed. These are likely located at your installation destination of Java. These source files will be used to gather javadoc for the informal specifications.

There should be a zip file names ‘deps.zip’ in the src directory which will need to be extracted and all files added to the build path.

Other downloads should not be necessary as our project is setup using Maven. All other dependencies such as the Stanford NLP should already be included as dependencies for Maven to retrieve.

Initial Setup:

There are 3 packages we need to be concerned with for the basic setup steps. These packages are ‘parsing’, ‘treeMatching’, and ‘analysis’.

The ‘parsing’ package contains the code that is used to parse out our javadoc. Changes shouldn’t need to be made to this package.

Next the ‘treeMatching’ package houses classes to handle our parse tree matching analysis. This package can remain unchanged as well.

Lastly the ‘analysis’ package is where rule files are inserted as well as our file paths for our tree matching and spec analysis. There are already some rule files pre-defined and referenced in the ‘ExceptionParseAnalysis’ class (which are added as arguments to the ‘CoreMapExpressionExtractor’ constructor) however if you wish to add more you should do so here. The ‘ExceptionParseAnalysis’ class also contains the main method which runs the analysis. You must provide a valid file path to the source code here. Also the ‘ExceptionDocStructureAnalysis’ class must be provided a file path just like the previous class.

Running the code:

Running the code is simple. Either go to ‘ExceptionDocStructureAnalysis’ class and run it for tree matching or go to the ‘ExceptionParseAnalysis’ class and run it for JML generation. At the moment generated JML is written to a file “results.txt” however the output/use will be different for each user.

Appendix II: Alternative Approaches

Social Approach:

The original project explained to our team by Professor Rajan is vastly different than the project we ended up delivering. We originally were hoping for more of a social approach (hence the project name). The following is the general idea behind our original approach:

Idea:

Create a socio-technical solution that will leverage the success of community collaboration platforms such as StackOverflow. Create specification for trivial methods automatically using code analysis techniques (already done by other people). Post questions about specifications on StackOverflow to receive answers. Parse and analyze these answers and store the solutions for others to use. Once we have established an acceptable level of interest on StackOverflow we then planned to merge our project with May1639's project which was essentially a StackOverflow clone (with added features) made specifically for formal specifications.

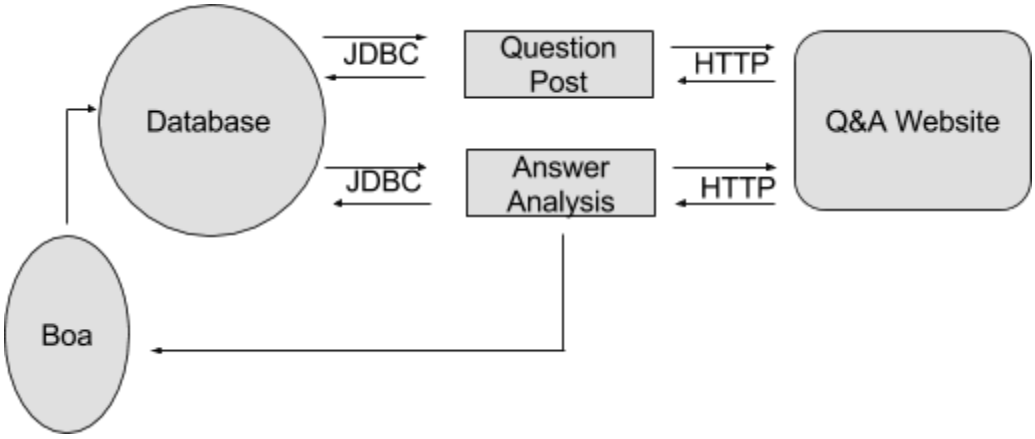


Figure 3: System diagram of our original system

Risks:

We knew going into this approach that there were significant risks. The most prominent risk was that StackOverflow users would not want to (or be able to) answer our

questions. We also knew that using the StackOverflow API for auto-posting questions was a gray-area as far as terms and conditions.

Process:

Our process was fairly straightforward for this approach. We created a module for posting questions and a module for retrieving questions after they had been answered. To get a question to post about we retrieved a method from Boa (a large scale software repository). We would then post a question that was generated based on the method and store relevant information (the question ID) into our own database so we could later retrieve it.

Shortcomings:

We had initially hoped that users would be more receptive to giving proper JML when asked about specifications. It turned out that most users did not know anything about JML so they were unable to answer our questions. This forced us to ask questions in a much less structured fashion. This did get us some better responses but overall the responses were fairly low quality and it was very common for users to simply tell us to look at the documentation for our answer. Eventually our StackOverflow account was banned due to the frequent and low quality posting.

Our Response:

As mentioned before we were often told to just look at the documentation. Although this was not the response we were hoping for we realized that there was some truth to it. We were not getting the structured responses that we had hoped for and now for a user to answer our question they were essentially just copying information from the javadoc. So we decided to take a new approach and cut out the middle man by directly using the javadoc of the JDK. This eventually led us to our current approach.

Appendix III: Other Considerations

anyCode:

An interesting project that we researched a bit in between moving from StackOverflow to the javadoc was anyCode. anyCode is a tool that takes natural language and turns it into actual java code. For example the phrase "*make a new file*" would turn into "*File file = new File();*". Our client was hoping that we would be able to use some techniques that anyCode used (or tweak anyCode itself) in order to analyze informal specifications. Unfortunately the code was extremely difficult to work with and we eventually realized that the process relied too heavily on "learning" by looking at java code making it much less useful for us. Luckily it was around this time that we made some headway with the Stanford NLP and were able to continue in that direction.

Our Takeaway:

Overall the problem presented to us turned out to be significantly more difficult than we had originally anticipated. Learning how to use the Stanford NLP was tough but rewarding. Despite the difficult workload for a 2-member group and some demoralizing setbacks that we had we feel that we were still able to produce a useful piece of software.