# **Source Code Feature Extraction for Automated Software Engineering**

**Final Project Report**

**Final Requirements, Design,**

**Implementation & Testing**

### **IBM - Dr. Jamie Jennings**

**CSC 492 15:**

**Jordan Connor**

**Trevor Gasdaska**

**Spatika Ganesh**

**Maya Shankar**

**North Carolina State University**

**Department of Computer Science**

**May 9, 2017**

### Executive Summary

(Trevor Gasdaska, Spatika Ganesh)

Brown, Notzli, and Engler argue in their paper “How to Build Static Checking Systems Using Orders of Magnitude Less Code” that fully parsing source code is unnecessary for conducting static analysis [1]. Instead, they suggest building “micro-grammars”: expressions that describe only portions of a language relevant to the checker. By analyzing only the areas of interest in the source code, useful insights can be made about the code without the overhead of processing unnecessary information. Once the information has been extracted from the source code, it can be used in a variety of applications including, but not limited to, those from the field of Automated Software Engineering.

The Rosie Pattern Language (RPL) or colloquially called “Rosie” is a lightweight, scalable language used for parsing expression grammars and shares concepts and notation with regular expressions. It excels at making the “micro-grammars” mentioned above. As RPL is designed like a programming language, it makes creating, maintaining, and sharing patterns easy.

Our project, under the working title Source Code Feature Extraction (SCFE), is to create a library of micro-grammars for RPL that is able parse a set of popular languages for a subset of interesting and useful features. We developed these micro-grammars following an iterative process which allowed us to develop good patterns quickly. Our project is developed on GitHub and is licensed under the MIT license. These patterns have been tested using hand-curated, valid source files either created by us or found on GitHub. Any test files found through GitHub are licensed under the MIT license.

We have conducted language-usage research which informed our decision on the languages we have parsed, including the popular languages Java, C, C++, Python, etc. and some unique languages such as Bash. We worked with our sponsor, Dr. Jennings, to select the set of features to look for in each language, studied RPL and walked through the process of creating a pattern for Java line comments. Additionally, we have used what we learned from our sponsor meetings to refine our process. By starting each iteration with a research focused phase and then transitioning to a iterative loop of developing and testing patterns while working as a team we made the most of our teams resources and time.

### Project Description

#### Sponsor Background:

(Maya Shankar, Jordan Connor)

IBM has historically and continues to be a primary leader in the technology industry. Continuing this trend with the growth of big data analytics IBM has been one of the forefront pioneers in the quickly growing industry. As this industry is still in its infancy IBM has quickly given focus to the invention and development of the necessary tools and technology needed to perform such analytics. In this process they have attracted many bright individuals who in their own right are pioneering the field. Dr. Jamie Jennings is one such pioneer and the creator and author of the Rosie Pattern Language (RPL), a lightweight scalable alternative to regular expressions.

Sponsor Info:

Jamie Jennings, Ph. D.

3039 E Cornwallis Rd, Research Triangle Park, NC 27709

(919) 543-0123

jjennings@us.ibm.com

#### Rosie Pattern Language:

The Rosie Pattern Language (RPL) is a scalable and lightweight language used to parse expressions in languages, similar to what regular expressions would be used for. It is easy to create and maintain patterns in RRL. These patterns can be used by themselves or in combination to create a tree-like structure of different features in source code written in different languages. To run Rosie, run the following commands:

git clone ‘<http://github.com/jamiejennings/rosie-pattern-language>’

cd rosie-pattern-language

make

make install

To test and understand how Rosie works, refer to the patterns in the /rosie-pattern-language/rpl directory. There exists a read-eval-print loop (repl) command line option through which the pattern creation using rosie can be understood.

./bin/rosie -repl command opens the interactive read/eval/print loop that allows the creation of patterns and matching against arbitrary input specified. For more detailed information, refer to the documentation provided in the RPL Github page.

#### Problem Statement:

(Maya Shankar, Jordan Connor)

Traditional static analysis requires a full parser in order to parse the necessary details of a source file. This typically creates large technical overhead as the parser must be developed and well tested. However, recent study in micro-grammars suggest that we can perform the same analysis without examining the entirety of the source file. These micro-grammars allow parsers to be built that only examine and retrieve the parts of a program which are of interest to the analyzer. Although the RPL was designed to act as the parser in such a process, currently useful micro-grammars are not available out of the box. The language would thus benefit greatly from a library of useful micro-grammars capable of parsing a set of interesting and useful features given source files from a set of popular programming languages.

#### Project Goals & Benefits:

(Spatika Ganesh, Jordan Connor)

The overarching goal of this project is to create a library of micro-grammars that identifies and parses a set of various features across different languages. These features will represent common syntactic paradigms in programming languages such as loop definitions, method/function definitions, and dependency statements. The languages chosen will represent a spread of the currently most popular and long standing languages such as Java, C, C++, and Javascript. This library will benefit RPL users by providing base examples of patterns for future users and developers. However, more directly in terms of our sponsor the library will be directly integrated into there DevOps process. Currently their analytic process must pull raw source files and fully parse such files to gain necessary details needed for purposeful analytics. However, as seen in Fig.1 SCFE will act as a filter between the raw source repository and the analytic process. This will allow IBM analysts the ability to focus on the study of raw data and not the parsing process. However, at the time of this report the actual extent to SCFE’s use in this analytics process is proprietary and confidential to IBM.

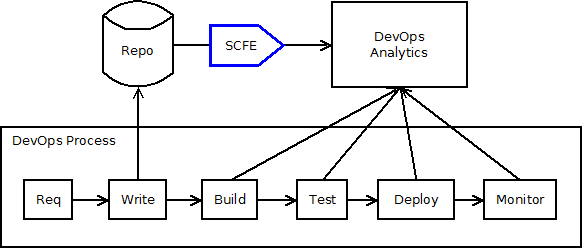


Fig 1. Big Picture (Trevor Gasdaska)

#### Development Methodology:

(Maya Shankar, Jordan Connor)

The initial portion of the project was dedicated to the research of languages and features in order to identify the patterns that will be contained within the library. Table 1 represents this set of elicited patterns in what will hereafter be referred to as the requirements matrix. Each entry in the matrix represents a clear development goal and as such suggests a highly iterative Agile approach to development. In our process an ideal iteration represented one week starting every Thursday after our weekly sponsor meeting. During these weekly meetings we delivered the results of that weeks iteration to our sponsor and discussed any necessary changes that may have arised during development challenges. During each iteration we attempted deliver an entire column of the matrix going from left to right. However, as the project moved forward this became a difficult task to accomplish. At the end of the development cycle as shown in the requirements matrix we completed a total of 5 pattern creation iterations as opposed to the original goal of 8. Although it is important to note that we had 3 iterations cycles completely dedicated to testing and defect correction, which we determined necessary in order to provide a working product.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Comments | Dependencies | Function Definitions | Class/Struct Definitions | String Literals | Exception Handling | Function Bodies | Class Bodies |
| Java | X (ALL) | X (ALL) | X (M.S.) | X (M.S.) | X (J.C.) |  |  |  |
| C | X (ALL) | X (ALL) | X (S.G.) | X (S.G.) | X (J.C.) |  |  |  |
| C++ | X (ALL) | X (ALL) | X (J.C.) | X (J.C.) | X (J.C.) |  |  |  |
| C# | X (ALL) | X (T.G.) | X (T.G.) | X (T.G.) | X (T.G.) |  |  |  |
| Python | X (ALL) | X (J.C.) | X (J.C.) | X (J.C.) | X (J.C.) |  |  |  |
| JavaScript | X (ALL) | X (S.G.) | X (S.G.) | X (S.G.) | X (S.G.) |  |  |  |
| Ruby | X (ALL) | X (T.G.) | X (T.G.) | X (T.G.) | X (T.G.) |  |  |  |
| R | X (ALL) | X (M.S.) | X (M.S.) | X (M.S.) | X (T.G.) |  |  |  |
| Go | X (ALL) | X (M.S.) | X (M.S.) | X (M.S.) | X (T.G.) |  |  |  |
| Bash | X (ALL) | N/A | X (J.C.) | N/A | X (T.G.) |  |  |  |
| VB | X (ALL) | X (J.C.) | X (T.G.) | X (T.G.) | X (T.G.) |  |  |  |

Table 1. The Requirements Matrix. An X represents an implemented requirement (Trevor Gasdaska, All)

#### Challenges:

(Maya Shankar)

The largest challenge of the project was the learning curve of RPL. However, this was mitigated greatly thanks to Dr. Jennings as she graciously dedicated the time to personally teach us the language. The existing documentation of RPL was also of great help.

We also faced challenges with certain quirks of the chosen languages. For example, although Bash does have dependencies through explicit program calls it is currently not feasible to parse these dependencies using RPL. Another example is not being able to parse specific code blocks in Python due to its implicitly defined syntax. Issues of this nature were removed from our requirements because they were determined to infeasibility. Other oddities were tracked using GitHub issues and corrected as necessary.

### Resources Needed

(Trevor Gasdaska)

|  |  |  |  |
| --- | --- | --- | --- |
| **Resource** | **Reason Needed** | **Status** | **Acquisition Info** |
| Rosie Pattern Language | Rosie is the language we are writing patterns for | Acquired | https://github.com/jamiejennings/rosie-pattern-language |
| GitHub | We will be using GitHub as our source control | Acquired | GitHub.com |
| Travis CI | We will be using Travis to run our regression testing suite | Acquired | https://travis-ci.org/ |
| Python | Our regression testing is run using a python script | Acquired | https://www.python.org/ |
| Source Files to use as Test Cases | We want to test our patterns on production code | Acquired | Found on GitHub with an MIT license or created by hand |

Table 2. Resources Needed (Trevor Gasdaska)

### 

### 

### Requirements

(Spatika Ganesh, Jordon Connor)

#### Overall View

The overarching goal of this project was to design and implement a family of micro-grammars that facilitates parsing of source code in various languages. In general these micro-grammars will represent an abstract set that can be used in any application that may require parsing of source code. For instance, IBM will specifically be using the patterns to parse data that may or may not be of interest to their pre-existing static analysis processes. The elicited set of patterns has been represented in the form of the matrix found in Table 1.

#### Functional Requirements

1. The system shall maintain a pattern in the Rosie Pattern Language that can match zero to many occurrences for a given entry in the requirements matrix (Table 1)
2. The system shall maintain a set of curated input source files and verified output files as generated by Rosie to be used as an example.
3. The system shall support introduction of new patterns at any point of development.

#### Non-Functional Requirements

1. The system shall not use any maintained source code for purposes other than matching patterns.

#### Constraints

#### Language

#### All patterns must be developed using Rosie Pattern Language (\*.rpl files)

1. The system shall be maintained and available on GitHub under the MIT license.

#### Testing

1. The system shall use Travis CI to maintain automated regression testing during integration.
2. The system uses python scripts to test the patterns.
3. The system shall use source code publicly available on GitHub licensed under the MIT license for testing purposes.
4. The source code used shall compile.
5. The source code used should be a language defined in the requirements matrix.

### Design

(Trevor Gasdaska)

### 

This project consists of a set of micro-grammars written in the RPL which can extract various features from source code written in several of the most popular languages (Table 1). Source files are run through Rosie and matched against specified patterns. Rosie then outputs a hierarchical JSON encoded output of the extracted feature.

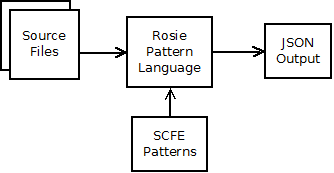


Fig 2. The SCFE project structure.

#### Structure of the JSON Output

The structure of the JSON output is determined by the way the pattern is written. For example consider a pattern, dependency\_stmnt, for matching Java import statements:

import = “import “

dependency = {!”!” .}\*

stmnt\_end = “;”

dependency\_stmnt = {import dependency stmnt\_end}

We see that the dependency pattern contains three subpatterns, java.import, java.dependency, and, stmnt\_end. Rosie will give insight into these three subpatterns in the JSON output. The output for the above pattern matched on “import java.util.\*;” is as follows:

{"dependency\_stmnt":

{"text": "import java.util.\*;",

"subs":

[{"import":

{"text": "import ",

"pos": 1.0}},

{"dependency":

{"text": "java.util.\*",

"pos": 8.0}},

{"stmnt\_end":

{"text": ";",

"pos": 19.0}}],

"pos": 1.0}}

As seen above, the output is structured as a tree, with the root being dependency\_stmnt and the child nodes being: java.import, java.dependency, and, stmnt\_end. Rosie also includes the complete text of what was matched along with the position in the input it was found.

Rosie gives control to the pattern writer to decide what shows up in the output via the the alias keyword. Including the keyword in a sub-pattern declaration informs Rosie to not include that sub-pattern in the parent-pattern’s output. This keyword can be added to the Java import pattern as follows:

alias import = “import “

dependency = {!”!” .}\*

alias stmnt\_end = “;”

dependency\_stmnt = {java.import java.dependency stmnt\_end}

When the same input “import java.util.\*;” is matched against the updated pattern, the output is structured as follows:

{"dependency\_stmnt":

{"text": "import java.util.\*;",

"subs":

[

{"dependency":

{"text": "java.util.\*",

"pos": 8.0}}}

],

"pos": 1.0}}

Since the schema of the JSON output is determined by the way the pattern is written, prior to developing each pattern, the team must decide on the most intuitive output schema for the particular language, feature pair and design the pattern accordingly.

#### Using the Language Feature Micro-Grammars

Features can be extracted from a given source file by running Rosie and providing the source file and desired pattern as command line arguments. Rosie by default pretty prints the JSON output in console. The full JSON structured output can be obtained by including the -encode json flag.

Once the JSON output is obtained the end user can use it as they see fit. For example, a user could create a dependency graph of a Java project using the dependency\_stmnt pattern defined in the previous section by parsing the JSON output and extracting the dependency’s. The extracted dependencies could then be associated with the file they were extracted from giving a single level dependency structure. Running this on an entire project and combining the results would produce a full dependency graph.

### 

### 

### Pattern Creation Process

(Trevor Gasdaska)

The pattern creation process is highly iterative and fueled by teaming (see [Fig. 3]). We developed the process while walking through creating our first pattern for Java line comments, taking careful note of the sticking points and successes. The factors we found to be most influential in developing a pattern are teamwork, iteration, having good test cases, and understanding the language-feature pair with the expected output.

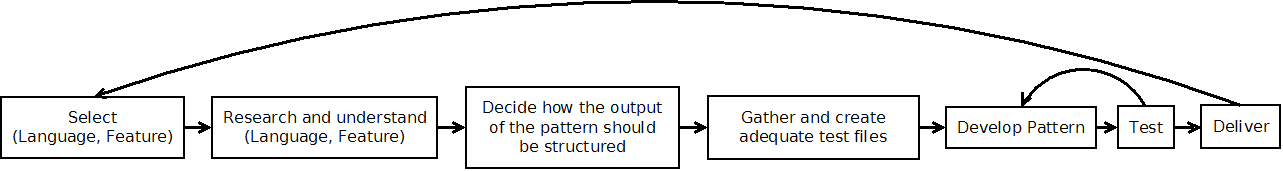


Fig 3. Pattern Creation Process

The process starts by selecting a language and a feature to extract. Once a language-feature pair has been identified, we then research the language syntax for that particular feature. This can be accomplished by looking up language documentation and should not be accomplished using experience alone to avoid missed edge cases or false assumptions. Next, we decide on the structure of the Rosie output as it can influence the test cases. After adequate research is performed and the structure of the output is finalized, test files will be gathered or created. Once the test cases are set up, a sub-iterative process of develop, test, and repeat starts. Working in pairs while developing the patterns increase the end quality of the patterns and the speed with which they are created. When the test cases pass, the pattern can be pushed to the repository and the process is repeated.

#### Process Example

###### Select (Language, Feature):

For this example we will be extracting dependencies from Java source code.

###### Research and Understand (Language, Feature):

Dependencies in Java begin with the keyword import followed by the package name and end with a “;”. The optional keyword static can be used between import and the package name [2].

###### Decide on JSON Output Schema

###### The pattern should extract dependency statements starting from the import up to and including the “;”. For Java dependencies the most important piece of information to extract is the name of the dependency. The keywords import and static and the “;” character should not show up in the output outside of the whole captured text as they aren’t as useful as the dependency name and can be extracted from the matched text if needed. Thus the output should contain a list of dependency statements which each in turn include the name of the corresponding dependency.

{

“dependencies”:

{

“subs”:[

{

"dependency\_stmnt":

{

"subs":[

{

"dependency":

{

"text":"...",

"pos":###

}

}],

"text":"...”,

“pos”:###

},

…

],

“text”: “...”

“pos”:###

}

}

###### Gather and/or Create Adequate Test Files

A successful pattern will successfully extract import statements in the following cases:

* Standard Import Statement: import <package-name>;\n
* Static Import: import static <package-name>;\n
* Multiple Statements on the same line:

import <package-name>;import <package-name>;\n

Thus a good test file would start with:

...  
import java.security.MessageDigest;  
import java.security.SecureRandom;  
import javax.annotation.Nonnull;import org.apache.commons.lang.StringUtils;import org.kohsuke.accmod.Restricted;  
import org.kohsuke.accmod.restrictions.NoExternalUse;  
  
import static com.google.common.base.Preconditions.\*;

...

Test files can be found on GitHub with an MIT license, created by hand, or found and then modified to cover edge cases.

###### Develop/Test Loop

An initial Java dependency pattern was developed in the previous section and provides a good starting point.

alias import = “import “

dependency = {!”!” .}\*

alias stmnt\_end = “;”

dependency\_stmnt = {java.import java.dependency stmnt\_end}

Rosie has two options for how it will try to match a given pattern against input. By default it tries to match the input line-by-line. However, some language features span multiple lines and as such require more than a single line to properly match. For this scenario, Rosie has a -wholefile flag which will bring the entire file into memory and attempt to match the entire file against the given pattern. For consistency in running our patterns, all patterns will be run with -wholefile enabled.

Running the above dependency\_stmnt pattern with the whole file flag enabled fails to match anything as the pattern will only match a single import statement.

Working as a team to troubleshoot how to extract single lines with the -wholefile flag enabled produces the second iteration of the dependency\_stmnt pattern:

alias import = "import "

alias stmnt\_end = “;”

alias line\_end = “\n”  
dependency = {!stmnt\_end !line\_end .}\*  
dependency\_stmnt = {import dependency}  
alias dependencies\_pre = {!import !line\_end .}\*  
  
dependencies = {{dependency / dependencies\_pre} {statement\_end / line\_end}}\*

The new pattern dependencies successfully matches the entire file. dependencies\_pre captures text that is not a dependency and is ignored in the output due to the alias keyword. This new pattern passes every test case except the one around static imports since the static keyword appears in the dependency text. Again, working as a team we find that modifying the dependency pattern as follows should remove the static from the output.

alias static = “static “

dependency\_stmnt = {import static? dependency}

Testing this one final time reveals that the modified pattern successfully passes all our test cases.

###### Deliver

The dependency pattern is now ready to be pushed to the GitHub along with the test files and the manually verified output JSON file.

### 

### Implementation

(Jordan Connor)

#### Git Branching Strategy

For our branching strategy we created a single branch for every language in requirements matrix apart from the master branch and also maintained a development branch. The idea being that at any given time only a single developer should be working on a specific language and as such branches should not be subject to merge conflicts. These branches were kept in sync with the primary development branch, which was pulled into master at the end of every iteration.

#### Source Code Directory Structure

In the discussion of the directory structure of the project we will use “senior-design-rpl” to represent the root of the project.

The SCFE library is currently located in the directory “senior-design-rpl/langlib/”. This directory contains an rpl file defined for every language found in the requirements matrix. These files are named using the languages most common abbreviation or fully expressed name depending on ambiguity. A rpl file containing patterns common to all language files is also found under “lang-common.rpl”.

Test files for a given language and feature along with their respective output are found in the following directories,

Input: “senior-design-rpl/testfiles/<language>/input/<feature>/<feature>.<langauge\_suffix>

Output:

“senior-design-rpl/testfiles/<language>/output/<feature>/<feature>.json

These directories are obviously parallel in nature in order to ease testing processes i.e. switching input -> output gives the output of a test based off feature and language names alone. It is important to also note the output files are specifically manually verified output of patterns.

#### Travis CI and Automated Testing Implementation

A regression test script has been written in Python. This script will execute Rosie on any source file found in an input directory of a feature and language combination. If this file has a matching output file the resultant output of Rosie is directly compared and any difference is considered a regression. This regression test was executed by a Travis CI implementation configured in “senior-design-rpl/.travis.yml”. Specifically Travis CI launches the regression test on an Ubuntu image given a push to any branch. The “build” is considered a failure if Rosie could not be built, if the regression script could not be executed, or if the regression script reported a regression.

#### Rosie Configuration

Rosie uses a manifest file to handle importation of rpl files. A pre-configured manifest exists at “senior-design-rpl/MANIFEST” that will allow any user of Rosie to clone our repository and immediately access our library through a simple Rosie command line option: “-manifest <manifest file path>”.

#### Iteration 1

Iteration 1 entailed the delivery of the first column of the requirements matrix: “comments”. This iteration was successfully delivered, but resulted in the creation of GitHub issues 1 and 2. However, the “block comments” pattern for Python was not delivered as it was discovered that Python does not have well defined block comments.

#### Iteration 2

Iteration 2 entailed the delivery of the second column of the requirements matrix: “dependencies”. This iteration was successfully delivered save for the implementation of the bash language pattern. Bash does not have traditional dependencies and upon discussion with the project sponsor it was decided to delay the development of this pattern. GitHub issues 3, 4, 5, and 6 were created during this iteration.

### Iteration 3

Upon discussion with the project sponsor iteration 3 was decided to be a “testing” iteration. This iteration was completed, and entailed the implementation of Travis CI, pattern documentation, and correction of all current GitHub issues (1-6).

### 

### 

### 

### Iteration 4

Iteration 4 originally entailed the delivery of the third column of the requirements matrix: “functions”. This iteration was not successfully delivered on time as our weekly sponsor meeting did not occurring on time. During this iteration we also ran into a large challenge dealing with recursive patterns in RPL, which hindered our initial pattern design.

### Iteration 5

Iteration 5 originally entailed the delivery of the third column of the requirements matrix: “classes”. This iteration was not successfully delivered on time. The recursion issues found in iteration 4 were discovered to be a generalized problem in features moving forward. During this iterations’ sponsor meeting, we restructured our requirements to add versions of patterns that had recursion implemented, and versions that did not. These new patterns are “Function Bodies” and “Class Bodies”, and were pushed towards the end of the development cycle.

### Iteration 6

Iteration 6 entailed the delivery of the third and fourth column of the requirements matrix: “function/classes”. Upon the discussions in iteration 5 we decided to focus on catching up to our initial timeline, and delivered the non-recursive version of the patterns.

### Iteration 7

Iteration 7 represented the second testing iteration. Here, we focused on adding more test cases for the already written patterns. We decided that each feature should have 3 distinct input test files, and worked to bring this requirement to all previous patterns.

### Iteration 8

Iteration 8 was the last iteration of the development cycle, and entailed the delivery of the fifth column of the matrix, and also continued testing of the previous columns in order to meet our new quality assurance requirements. This iteration was delivered during the final handoff of the project.

### Test Plan & Results

(Jordon Connor)

As stated in the requirements section, an important aspect of the project is verification of pattern output. However it is important to note that RPL is not a general purpose programming language and more importantly not Turing complete. Because of this, traditional software testing is not necessarily relevant to any patterns developed. For instance, code coverage although technically possible to track may simply be redundant data. This is because in context, it is exactly equivalent to the attempted amount of unique patterns matched during execution and as a test should generally test a singular pattern for accuracy, we as developers should simply hold responsibility in assuring every pattern is tested. With this in mind it is an extremely difficult task to determine whether or not a pattern is well tested. Therefore, a traditional test plan with traditional acceptance tests is not a viable option. Instead, a highly malleable and iterative testing process was used within our pattern creation process. During pattern creation, developers were required to research and provide example input files for the pattern being developed. These files were in total a combination of code specifically manufactured by the team and real example code found in open-source software licensed with reuse in mind. We then used the generated suite of input files to manually verify the output of the RPL pattern being developed. Once the output was verified to be correct, it along with the input files was added to an automated regression test suite. This process is outlined in [Fig. 4]. Travis CI was used to manage automated test execution while the testing scripts will be written in Python.

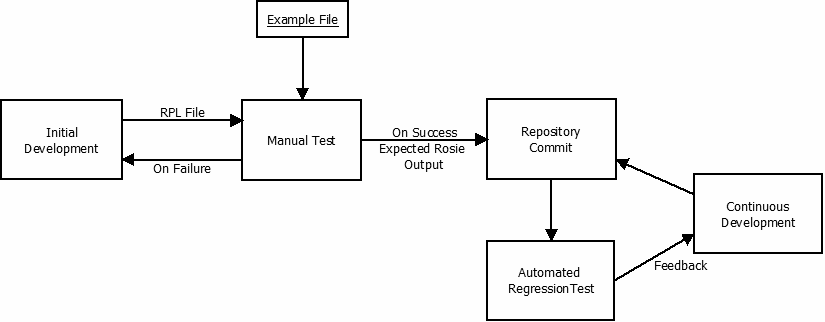


Fig 4. Testing Process

### Suggestions for Future Teams

(Trevor Gasdaska)

The future work for this project falls into three main categories: improving and expanding existing patterns, writing new patterns, and creating applications for the patterns.

We encountered a few challenges trying to extract some features from some languages. Python was a particularly challenging language to parse. We were unable to extract some python features such as function and class bodies because of the reliance on a variable amount of whitespace. In order to extract these features from arbitrary python source code, we require access to memory to remember how much white-space differentiates the bodies from the surrounding code. Dr. Jennings has suggested that future versions of Rosie may have the functionality to handle these types of situations. If this feature is realized, future teams could modify existing patterns to extract these types of features. Bash dependencies was also another feature that we found challenging to parse. Bash has dependencies in the sense of the other programs it calls, but lacks the dependency declarations found in other languages. We were unable to think of a good way to extract the dependencies. One suggestion we did not try was to keep a list of bash key-words and extract the other words as dependencies. This would most likely not be 100% accurate so future teams could research how to best extract these dependencies.

Our project focused on some of the most popular languages, and useful features. However, this list is far from exhaustive. Future teams could expand on this project by extracting features from new languages, and extracting new features from these languages. As far as new features to extract, future teams could investigate loops, conditionals, lambda functions, and annotations as these are features we thought would be useful, but were less important than the features we selected. The existing patterns could also be expanded to extract from new languages. One suggested language is PHP, as there was interest in being able to extract features from this language during our poster session.

Our project has a near endless set of possible applications. The big idea behind our project is the theory of micro-grammars, and that you do not need to fully parse source code in order to statically analyze it. Future teams could take the work that we have done and use a micro-grammar approach to create new static analysis tools that are more efficient and have the potential to be language agnostic. Potential things to analyze would be, building dependency graphs, security analysis, and error prediction.

In general, the most useful piece of advice we can give for any team picking up a project involving Rosie would be to spend a lot of time learning the language. Study our patterns and documentation, play around with the differences between including braces around statements and not using them, learn and understand how grammars can be used to parse more difficult patterns involving recursion, learn and utilize the wholefile option, and make the read-eval-print loop your best friend. We found there to be a very high ROI on investing time into developing a deep understanding of the Rosie language. Starting to write patterns too early may result in having to go back and rewrite those patterns later on.

**Team Contact Information**

Jordan Connor, Team Leader, jhconnor@ncsu.edu

Trevor Gasdaska, Planning Manager, trgasdas@ncsu.edu

Spatika Ganesh, Development Manager, sganesh3@ncsu.edu

Maya Shankar, Quality/Process Manager, mshanka@ncsu.edu

### 

### 

### 

### References

[1]: Brown, F., Nötzli, A., & Engler, D. (2016). How to Build Static Checking Systems Using Orders of Magnitude Less Code. ACM SIGOPS Operating Systems Review, 50(2), 143-157. doi:10.1145/2954680.2872364

[2]: Chapter 7. Packages. (n.d.). Retrieved March 03, 2017, from https://docs.oracle.com/javase/specs/jls/se8/html/jls-7.html#jls-7.5