FreeBSD:
Dependency Extraction

EECS 4314: Advanced Software Engineering | Assignment 3
March 24th, 2023

Authors: **Pink Fluffy Unicorns**

Benedict Miguel	216237364	benedictmiguel106@gmail.com
Jiachen Wang	215453988	reimsw98@my.yorku.ca
Keshav Gainda	216891319	gaindakeshav@gmail.com
Ketan Bhandari	217550765	ketan11@my.yorku.ca
Mohaimen Hassan	216187809	hassan08@my.yorku.ca
Pegah Fallah	216262081	fpegah@my.yorku.ca
Shami Sharma	217262510	shami012@my.yorku.ca
Sidharth Sudarsan	216697120	lensman@my.yorku.ca
Suryam Thaker	217281031	suryam@my.yorku.ca
Xiaochuan Wang	214107106	wangx997@my.yorku.ca

Table of Contents

Abstract

This study compares and analyzes three dependency extraction techniques: Understand, srcML, and the #include method. Performance of these techniques is judged in terms of precision, recall, and efficiency through quantitative and qualitative analysis. Quantitative analysis determined that Understand identifies more dependencies than srcML and #include-based extraction, though it may introduce false positives. Both srcML and #include-based extraction methods have inherent limitations, such as reliance on import/include statements and increased processing time, that can lead to missed dependencies. Qualitative analysis is necessary to rationalize the reasons behind these differences. Factors such as granularity of scanning, file types considered, and handling of architecture-dependent code contribute to the variance observed between the dependency extraction techniques. Precision and recall calculations were performed for Understand, srcML, and #include techniques to assess the quality of extraction techniques. Each tool had its limitations, including processing time and the ability to recognize deeper dependencies. The study concluded with a ranking of the techniques based on the number of dependencies extracted.

Dependency Extraction Techniques

Understand

Understand is a static code analysis tool that allows for the program to be checked for potential errors and other issues. In this project, this tool was used to gather insight on the dependencies between different segments of the source code. When touching upon the source code level dependencies, this software is able to extract function calls as well as variable access relations, e.g when when function y calls upon function x. This tool can also be used to identify control flow and data flow dependencies which is primarily used in order to get a better understanding on how different segments of the code interact with each other. Dependencies can be established through various means, such as import or include statements, inheritance, implementation, method calls, and object initialization.

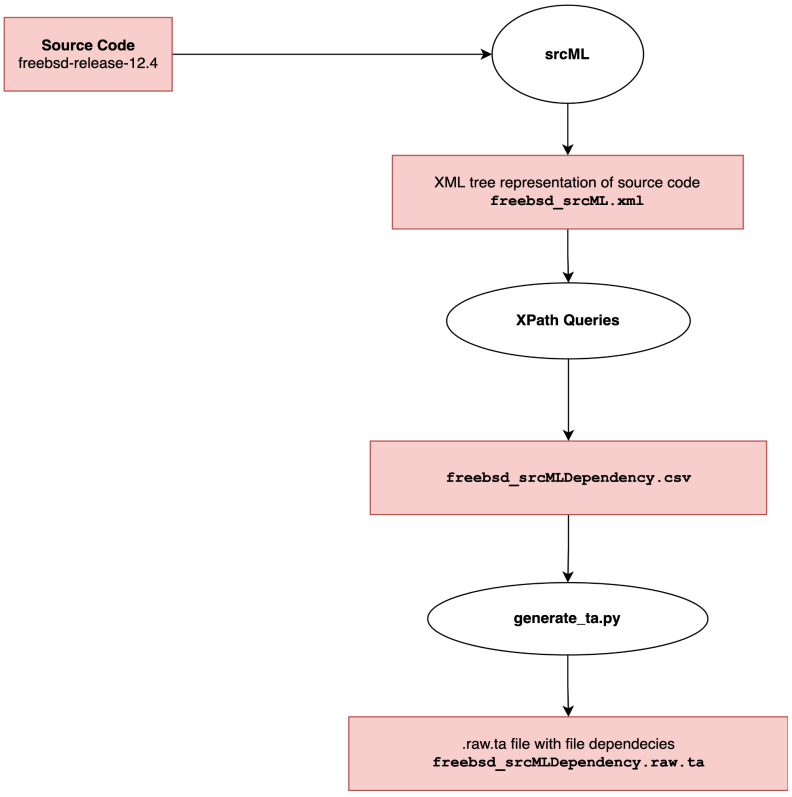

Figure 1: srcML Extraction Flow

srcML

The extraction technique of srcML is fairly simple but is computationally heavy. SrcML, given a source code, outputs an XML representation of the source code, where the

nodes of the XML file represent the different parts of the code. The source file itself is represented as a <unit> type of node where the attributes contain information regarding the filename, the language the source code file is written in and other relevant information. Because srcML converts source code into an aggregated XML file, we can use xpath queries to find every single file. Given the files, we then use xpath again to find the dependencies of the file by looking at what imports it uses by looking at the <include> nodes of the file. This extraction technique heavily relies on the import/include statements of the source code, in other words, if the dependency is not explicitly stated in the source code itself, or that it can't be represented in the generated XML file, that dependency will not be seen when extracting which may lead to a decreased number of dependencies extracted in comparison to the Understand dependency technique.

#include-based Extraction

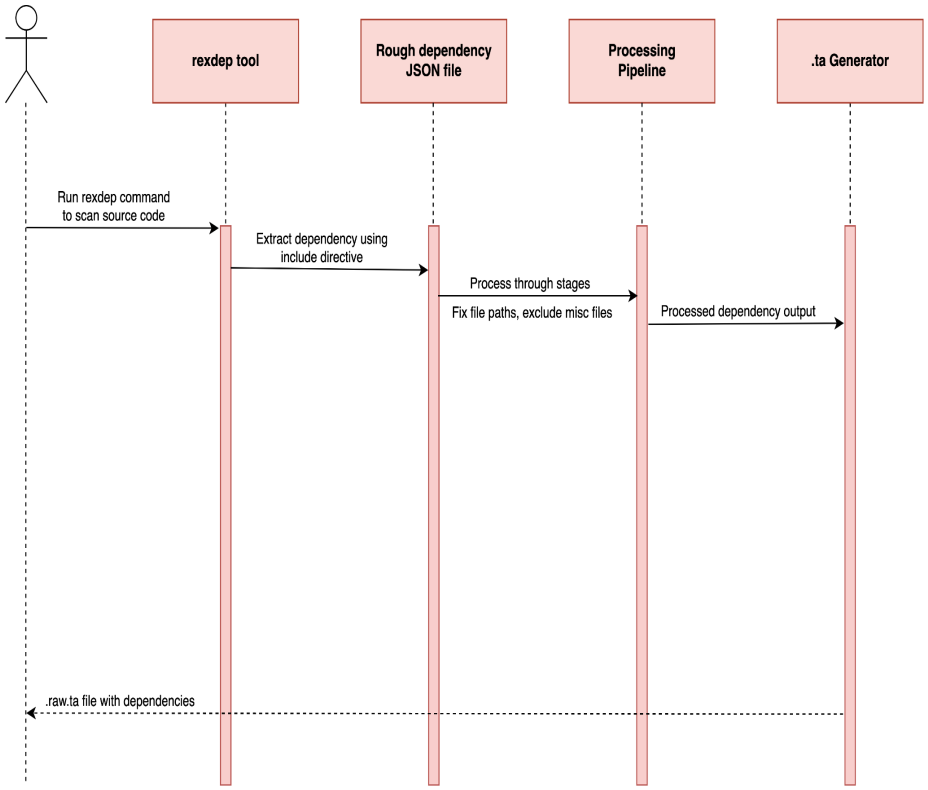

Figure 2: #include Extraction Sequence Diagram

Include-based Extraction is a technique that scans the source code and uses the include directive to find the dependency. The whole process can be like this, first running rexdep command to scan the source code, which is for finding the include directive and extracting the dependency. After this step we can have a rough dependency JSON file, and in this file we can not analyze the dependency yet, because this file has redundant code. So the next step we need to do is to process through stages and fix file paths, exclude misc files. This step will take a long time because we need to analyze each line. So, there will be a processing pipeline. After the processing finishes, we can get a ta file which contains the dependency of each subsystem.

Quantitative Analysis

Process

The first step in comparing and analyzing the differences and similarities among all three methods is to measure the number of instances and dependencies. We want to determine how many unique instances and dependencies each method has, as well as how many they share in common.

To accomplish this, we use a script to count the number of instances and dependencies in each ".ta" file generated by each extraction method. We begin by counting instances, where each instance is

located in a single file in the source code, and then measure the number of dependencies associated with each instance.

Since this process involves ".ta" files, instances are extracted from each line starting with "$INSTANCE", while dependencies are extracted from lines beginning with "cLinks". To count dependencies, if a file is the "from file", we add an edge from that file to the other file that provides functionalities, increasing the dependency count for that "from file" by one.

During comparisons, both instances and dependencies are iterated. Iterating instances is straightforward, but because different methods may generate .ta files in different orders, instances may not be inserted into dictionaries in the same way. Therefore, in the worst case, iterating a dictionary of instances and checking if each instance is in the other dictionary may take up to O(n^2) time based on file size.

However, extracting dependencies is less time-consuming and amortized. Since we only need to compare dependencies from the same instance each time, the comparison does not need to occur across the entire set of dependencies. Once we have data on the dependencies associated with each instance, we can count them all together and begin the analysis.

Understand vs srcML

Using the process outlined above, we found the numbers of instances and dependencies, categorizing them into unique and common groups. We then created Venn diagrams to present these quantitative data as shown below to compare Understand and srcML:

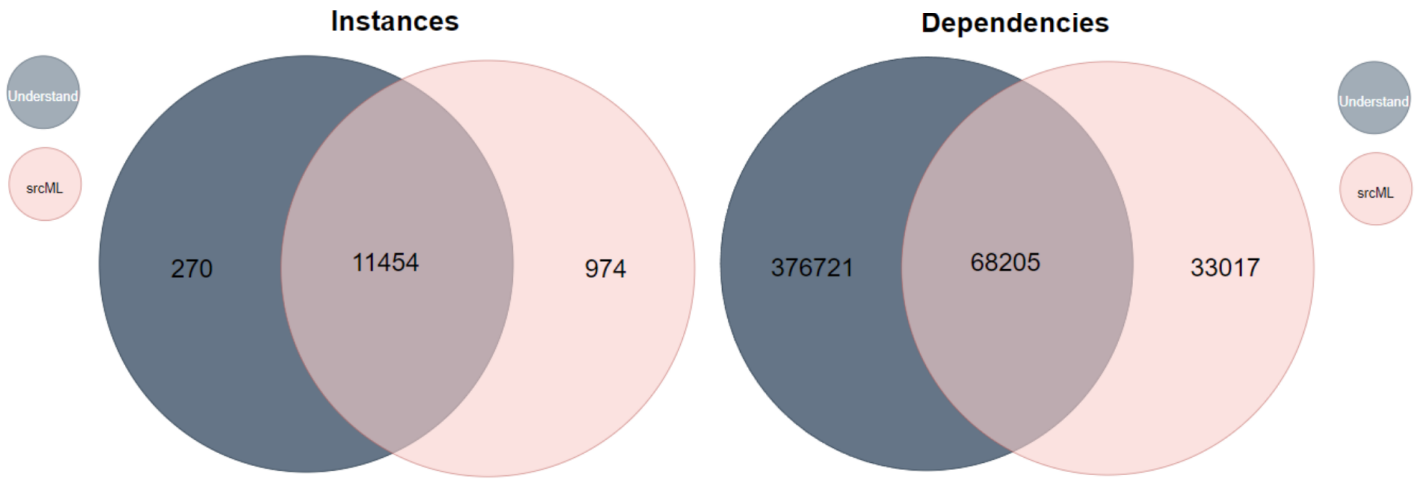

Figure 3: Venn Diagrams Comparing Understand and srcML

We can also create a table to list the data.

	Understand	srcML
Unique Instances	270	974
Common Instances	11454	
Unique Dependencies	376721	33017
Common Dependencies	68205	

Table 1: Statistics of Understand and srcML Comparison

A key comparison metric to look at is the number of unique dependencies identified by each tool. If we assume that the identified unique dependencies are all unique and correct, we can see that out of 477,943 identified dependencies, 409,738 of them are not identified by both tools at the same time. This is a significant number, as it shows that **85.5%** of the dependencies were not identified by both tools.

In the worst case, if we use srcML only, up to 376,721 dependencies would be missing. On the other hand, using Understand only would mean, up to 33,017 dependencies would remain unidentified.

We can infer that either Understand identified a lot of incorrect dependencies, or srcML missed a huge portion of them. This also shows the importance of using multiple tools for ensuring accuracy.

srcML vs #include

Similarly, we came up with the following Venn diagrams to visualize the instances and dependencies which are unique and common across the outputs of srcML and #include:

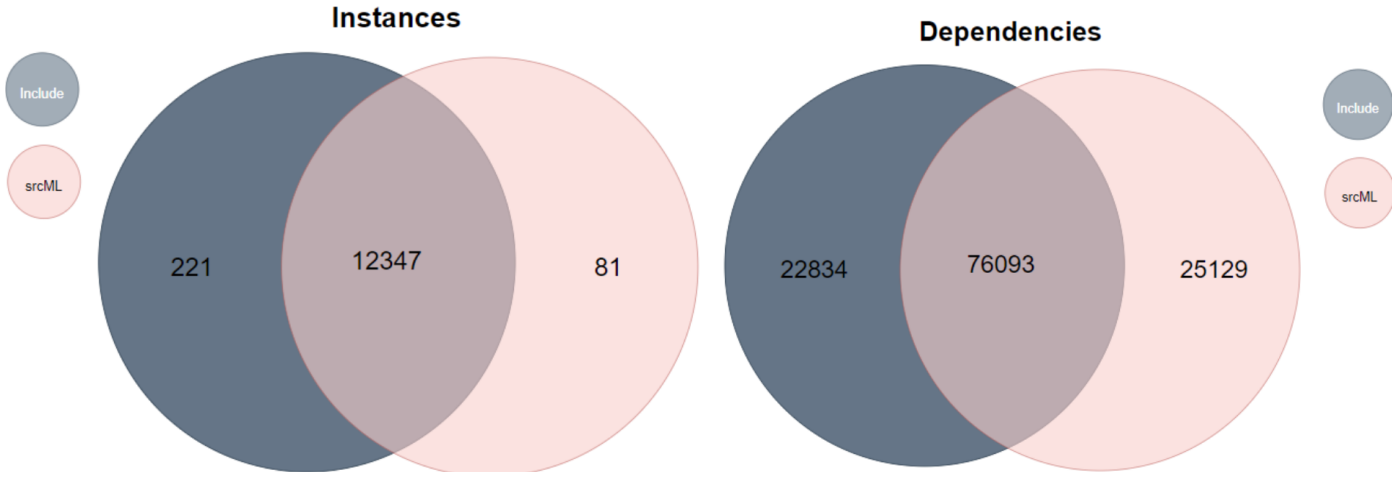

Figure 4: Venn Diagrams Comparing srcML and #include

From the diagrams, we can see the following information:

	srcML	#include
Unique Instances	81	221
Common Instances	12347	
Unique Dependencies	25129	22834
Common Dependencies	76093	

Table 2: Statistics of srcML and #include Comparison

Using the comparison metric as before, we assume that the identified unique dependencies are all unique and correct. We can see that out of 124,056 identified dependencies, 47,963 of them are not identified by both tools at the same time.

This shows that, in the worst case, if we use srcML only, up to 22,834 dependencies would be missing. On the other hand, using #include only would mean, up to 25,129 dependencies would remain unidentified.

#include vs Understand

Lastly, we compared the data for #include and Understand. The diagrams below show our findings:

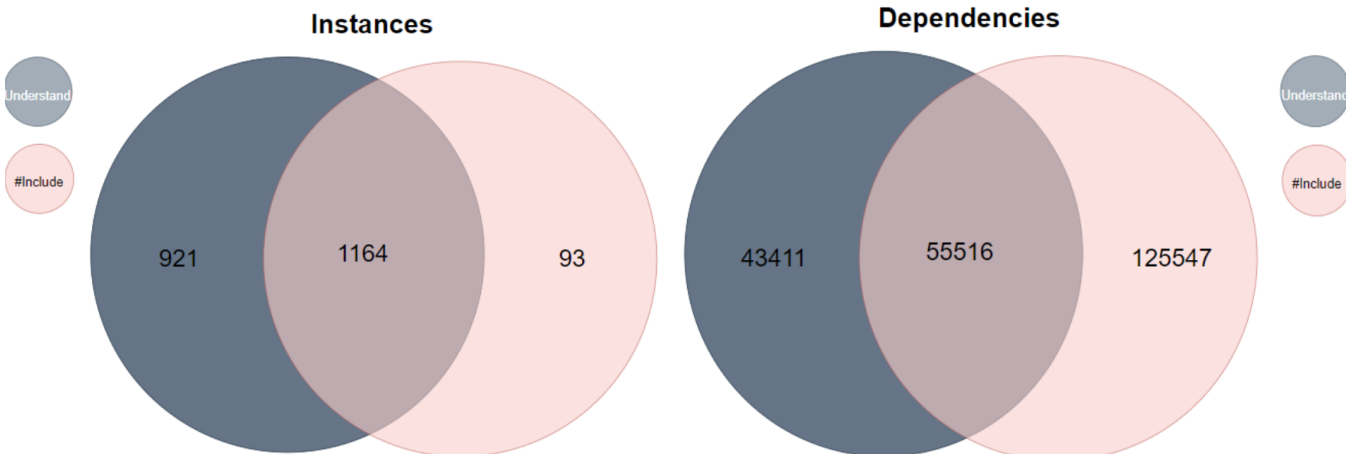

Figure 5: Venn Diagrams Comparing #include and Understand

From the diagrams, we can see the following information:

	#include	Understand
Unique Instances	93	921
Common Instances	1164	
Unique Dependencies	43411	125547
Common Dependencies	55516	

Table 3: Statistics of #include and Understand Comparison

Again, using the same comparison metric, we assume that the identified unique dependencies are all unique and correct. We can see that out of 224,474 identified dependencies, 168,958 of them are not identified by both tools at the same time. Similar to the case of Understand vs srcML, the latter number is quite significant - **75.3%** of the dependencies did not match across both tools.

This shows that, in the worst case, if we use #include only, up to 125,547 dependencies would be missing. On the other hand, using Understand only would mean, up to 43,411 dependencies would remain unidentified.

We can see somewhat of a trend when comparing a tool against Understand - we can infer that either Understand is providing an exhaustive list, which other tools are missing, or Understand is outputting a lot of false positives, meaning that it is introducing errors which other tools aren't.

Qualitative Analysis

Process

For the qualitative Analysis, we chose a stratified sampling method for the comparison process. In stratified sampling the data is partitioned into mutually disjoint subsets called strata, then randomly sample data from each stratum. It can be used to determine the number of individuals that need to be interviewed to obtain the result that accurately reflects the target population, as well as the level of precision in an existing sample. Since the given data was already divided into three subpopulations(i.e common to both A and B, unique to file A and unique to file B), therefore this stratified sampling method was deemed appropriate. In order to find the sample size we used a sampling calculator which is provided by the creative research system, with a confidence level of 95% and a confidence interval of +/-5%.

#include vs Understand

Following the research process described, the instances and dependencies were identified, and categorized into unique and common groups. Venn diagrams were then constructed to visually represent this data:

Total Population: 43,411 + 125,547 + 55,516 = 224,474

Based on the total population of 224,474, it was calculated that a sample size 384 is needed.

Subgroup	Equivalent Sample Size (Subpopulation / Total Population)	Percentage of Total Sample Size
Common Dependencies	(55,516/224,474) * 384 = 95 cases	(95/384) * 100 = ~24.7%
Unique to Understand	(43,411/224,474) * 384 = 74 cases	(74/384) * 100 = ~19.3%
Unique to #include	(125,547/224,474) * 384 = 215 cases	(215/384) * 100 = ~56%

Table 4: Equivalent Sample Size calculations for #include and Understand

Differences and Rationale

The distribution of dependencies in the total sample size indicates that approximately 24.7% are common dependencies, while 19.3% are unique to Understand and 56% are unique to the #include dependency extraction method.

One factor contributing to the differences is the granularity of scanning. Understand comprehensively scans and analyzes source code at a lower level, providing more detailed and accurate dependency information. On the other hand, the #include method derives dependencies purely based on #include directives, which may not capture all relations within the code.

Next, the file types considered during the extraction process is a factor that explains these differences. Understand includes *.c, *.h, and *.m files, whereas the #include method considers all extensions mentioned in the #include directives, such as *.c, *.h, *.m, *.S, *.inc, and *.dts. The broader range of file types in the #include method may contribute to the higher number of unique dependencies.

Lastly, the handling of architecture-dependent code also influences the differences observed between the two techniques. Understand accurately extracts relations for architecture-dependent code, whereas the #include method may encounter issues when dealing with expressions like "machine/../xxx," instead of the exact directory names.

Understand vs srcML

Total Population: 376,721 + 68,205 + 33,017 = 477,943

Based on a total population of 477,943, a sample size of 384 was determined to be appropriate.

Subgroup	Equivalent Sample Size (Subpopulation / Total Population)	Percentage of Total Sample Size
Common Dependencies	(68,205/477,943) * 384 = 55 cases	(55/384) * 100 = ~14.3%
Unique to Understand	(376,721/477,943) * 384 = 303 cases	(303/384) * 100 = ~78.8%
Unique to srcML	(33,017/477,943) * 384 = 27 cases	(215/384) * 100 = ~7%

Table 5: Equivalent Sample Size calculations for Understand and srcML

Differences and Rationale

Building upon the previously discussed factors, the differences between the Understand and srcML dependency extraction techniques can be further analyzed. Within this sample, approximately 14.3% are common dependencies between the two extraction methods, 78.8% are unique to Understand, and 7% are unique to srcML.

Granularity of scanning, file types considered, and handling of architecture-dependent code are once again key factors contributing to these differences. srcML derives dependencies purely based on #include directives in contrast to Understand which analyzes source code. In terms of file types, srcML considers all extensions mentioned in #include, *.c, *.h, *.m, *.S, *.inc, and *.dts. Lastly, in handling architecture-dependent code, srcML also uses expressions like “machine/../xxx” instead of the exact directory.

srcML vs #include

Total Population: 22,834 + 76,093 + 25,129 = 124,056

Based on a total population of 124,056 dependencies, a sample size of 383 was needed.

Subgroup	Equivalent Sample Size (Subpopulation / Total Population)	Percentage of Total Sample Size
Common Dependencies	(76,093/124,056) * 383 = 236 cases	(236/383) * 100 = ~61.3%
Unique to #include	(22,834/124,056) * 383 = 71 cases	(71/383) * 100 = ~18.4%
Unique to srcML	(25,129/124,056) * 383 = 78 cases	(78/383) * 100 = ~20.3%

Table 6: Equivalent Sample Size calculations for srcML and #include

Differences and Rationale

Around 61.3% are common dependencies, 18.4% are unique to #include, and 20.3% are unique to srcML within this sample. Both #include and srcML techniques use #include directives as the source of truth for dependencies. #include solution performs post-processing to fix inconsistencies. The two differ when it comes to querying. #include uses Unix based querying (sed, awk, grep) while srcML uses XML-based queries (xPath) queries.

Performance Measurements

Precision & Recall

Precision measures the proportion of correctly identified items among the retrieved ones, while recall evaluates the proportion of relevant items retrieved out of the total relevant items.[5] The quality of the srcML and #Include dependency extraction techniques will be assessed. The sample for the following precision and recall calculations are gathered from the quantitative analysis tables.

Sample size: 384

TP = Dependencies present in both the sample and Understand results = 358

FP = Dependencies present in both the sample and srcML results = 82

FN = Dependencies present in the sample but not detected by Understand = 27

Precision = TP / (TP + FP) = 358 / (358 + 82) ≃ 0.813559 ≃ 81.36%

Recall = TP / (TP + FN) = 358 / (358 + 27) ≃ 0.929825 ≃ 92.98%

Sample size: 383

FP = Dependencies in the sample that are also identified by the #include method = 309

FN = Dependencies in the sample that are not identified by srcML = 71

TP = Dependencies in the sample that are also identified by srcML = 321

Precision = TP / (TP + FP) = 321 / (321 + 309) ≃ 0.509554 ≃ 50.96%

Recall = TP / (TP + FN) = 321 / (321 + 71) ≃ 0.818877 ≃ 81.89%

Sample size: 384

TP = Dependencies from Sample that are also in Understand = 169

FP = Dependencies from Sample that are also in include = 289

FN = Dependencies from Sample that are not present in Understand = 215

Precision = TP / (TP + FP) = 169 / (169 + 289) ≃ 0.368966 ≃ 36.90%

Recall = TP / (TP + FN) = 169 / (169 + 215) ≃ 0.440104 ≃ 44.01%

Potential Risks and Limitations

Understand

Even though understand is a powerful tool to analyze the dependency, there are still some limitations and risks for it. First the programming language limit, understand can only support c++ java and python some new language can not be supported by understand, second we can not know how does it access the code, in other words, we do not know how the understand' algorithm, so we can not know if it analyze the code accurate enough. Last thing is that we can not not analyze the whole system, we can only pick some sample file and analyze it, and the process will also take a very long time, so it is really hard to analyze the whole system, even if we put all the files one by one.

srcML

The current limitations of srcML is based on the fact that dependencies can only be extracted if it can be represented as an XML file. This means that if there are dependencies that are not represented in the code, or are not explicitly stated in the code, srcML will not be able to generate an XML file that represents these dependencies. This inability causes a decrease in the number of dependencies extracted when compared to other extraction techniques such as Understand. Further analysis in the differences between the Understand technique and srcML technique created a perfect case of the limitations of srcML. Understand is able to find the dependency between two different .c files, while srcML is limited to creating dependencies between only a .c file and a .h file. Other limitations also present in the computational requirement of running srcML. Running large source code takes srcML a

substantial amount of time to generate the XML file. Further usage of xpath to find the dependencies also takes a substantial amount. As an example, an XML file of size 1.4gbs would take a processing time of 6-7 to fully extract all the dependencies using xpath queries. This means that as the size of the source code increases, the viability of using srcML as a dependency extraction tool decreases.

#include

The #include can analyze the dependency according to the include.h, so there may be the situation that it can not recognize the dependency, for example it can not the deeper dependency like parent-children or function call.

Data Dictionary

srcML: srcML is a language-independent source code transformation tool that converts source code into an XML representation.[4]

XML (eXtensible Markup Language): XML is a markup language designed to store and transport data in a structured and human-readable format. It uses custom tags to define elements and organize data hierarchically.[6]

XPath: XPath is a query language used to navigate and select nodes in an XML document. It uses path expressions to access specific elements or attributes and can also perform simple computations on the data. [7]

TP (True Positive): A true positive is an outcome in which a test correctly identifies the presence of a specific condition, such as a positive result for a correctly identified dependency in code analysis.[5]

TN (True Negative): A true negative is an outcome in which a test correctly identifies the absence of a specific condition, such as a negative result for a correctly identified non-dependency in code analysis.[5]

FP (False Positive):A false positive is an outcome in which a test incorrectly identifies the presence of a specific condition, such as a positive result for an incorrectly identified dependency in code analysis.[5]

Lessons Learned

The lessons learnt during the completion of this assignment were that the process of extracting the dependencies from the source code differs in technique as well as the outputs in a different file type. For instance, Understand outputs a .csv file which makes manipulating these values very easy using Pandas or Excel. In contrast, The Include method uses .json files, which is relatively fast to process. However, scrML by-far takes the longest because of the XML file type. Using XPATH to process large XML files increases the processing time. The contents of the outputs resulting from these techniques may also vary. For example, in the case of srcML, it is not able to create a cLink between two different files (cLinks file1.c **file2.c**) because the source code XML is not able to represent the connection of the second file. Finally, when ranking the number of dependencies each technique uses, the ranking is as follows; Understand (15.9 mb), Include (11.6 mb) and srcML (8.9 mb).

Conclusion

In conclude, for this project we extract the source code and uses three different approaches to analyze the dependency of FreeBSD, first approach is understand and the second approach is #include and the third one is srcML, all of these tools get different numbers of dependencies, after comparing these numbers, we analyze these statistic in two different aspect, qualitative and quantitative, the quantitative contains the number of dependencies and the number of instances, the qualitative contains the percentage of total sample size. Then we find the limitation of these three approaches which is that it takes a long time for them to process the source code. And get the conclusion by ranking each technique by dependency numbers.

References

1. Creative Research Systems. (n.d.). *Sample Size Calculator - Confidence Level, Confidence Interval, Sample Size, Population Size, Relevant Population - Creative Research Systems*. Survey Software - The Survey System. Retrieved March 23, 2023, from https://www.surveysystem.com/sscalc.htm
2. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. (2008, November 26). W3C. Retrieved March 23, 2023, from https://www.w3.org/TR/REC-xml/
3. The FreeBSD Documentation Project. (2012, 01 01). *FreeBSD Architecture Handbook*. FreeBSD Documentation. Retrieved March 23, 2023, from https://docs.freebsd.org/en/books/arch-handbook/book/
4. srcML. (n.d.). *srcML*. srcML. Retrieved March 23, 2023, from https://www.srcml.org/
5. Wikipedia Foundation. (n.d.). *Precision and recall*. Wikipedia. Retrieved March 23, 2023, from https://en.wikipedia.org/wiki/Precision_and_recall
6. *XML Path Language (XPath) Version 1.0*. (1999, November 16). W3C. Retrieved March 23, 2023, from https://www.w3.org/TR/xpath-10/
7. *XPath*. (n.d.). Wikipedia. Retrieved March 23, 2023, from https://en.wikipedia.org/wiki/XPath