BluePrint for Understanding the Differences in Apache Flink's Dependency Extraction

Dependency Extraction
By: Analysis Paralysis

Authors:
Aisha Mahmood - aisha895@my.yorku.ca
Harsimran Saini - saini19@my.yorku.ca
Esha Vij - eshavij@my.yorku.ca
Harrish Elango - elangoha@my.yorku.ca
Adewusi Fashokun - korede@my.yorku.ca
Siddhanth Bakshi - sbakshi@my.yorku.ca

Abstract

This dependency extraction report provides a detailed description of the three distinct architectural dependency extraction techniques employed in software development. The version of Apache Flink in context is 1.17.1. The tools that we employed to extract the dependencies are the following: the Understand tool, writing a script that searches for “include” directives and srcML. The objective is to give a thorough grasp of the benefits and drawbacks of each strategy, providing information that can guide best practices in software architecture analysis. The output formatted from both techniques was tailored to conform to the TA format. In order to lay the groundwork for the ensuing quantitative and qualitative evaluations, the report starts out by quickly outlining the specifics of how Include and srcML were implemented. The report also documented how these processes were implemented through clear visualisations. A quantitative evaluation comprises several statistics, such as the total number of dependencies that are extracted from each method and the proportion of similarities between them. Performance measures like recall and precision are also provided in order to assess the correctness and dependability of the analysis. The report also mentions potential risks associated with the approaches. With this research, practitioners will have a solid foundation for making educated decisions on software architecture analysis by learning more about the advantages and disadvantages of various dependency extraction methodologies. The comparison study's findings form the basis for upcoming developments in dependency extraction techniques, which will eventually strengthen and improve software development processes.

Introduction and Overview

This report offers an in-depth exploration of three distinct techniques for extracting dependencies from the source code: Include directives, srcML, and Understand tools. The document systematically outlines the procedures involved in each technique, emphasising their unique approaches to uncovering dependencies within the software architecture. Furthermore, the report provides a thorough quantitative and qualitative analysis, including precision and recall metrics, facilitating a comprehensive comparison of the effectiveness of the three methods. There are practical use cases for Understand and srcML, showcasing the versatility of these tools in real-world scenarios. The report also delves into the lessons learned during the assignment, highlighting the acquired skills in comprehending large software systems. Notably, the document conscientiously addresses the limitations inherent in the reported findings, particularly in the context of the challenges faced during the comparison process. In conclusion, this report serves as a valuable resource for understanding various dependency extraction techniques, offering insights into their applications and limitations within the intricate landscape of Apache Flink source code analysis.

1- Dependency Analysis Using Understand

1.1 Understand Tool

SciTools has developed Understand, a powerful static code analysis tool tailored to enhance software development processes. Compatible with a variety of programming languages including C, C++, C#, Java, and Python, Understand provides valuable insights into program source code. This tool excels in revealing dependencies and relationships between components, shedding light on code complexity for developers. With robust features like code navigation capabilities, comprehensive metrics, and visualization tools, Understand facilitates a deeper understanding of project structures, aiding developers in writing improved code and identifying potential issues. Positioned as a solution for making code maintenance more straightforward and the overall development process more efficient, Understand emerges as a valuable asset in the realm of software engineering.

1.2 Observations from Understand

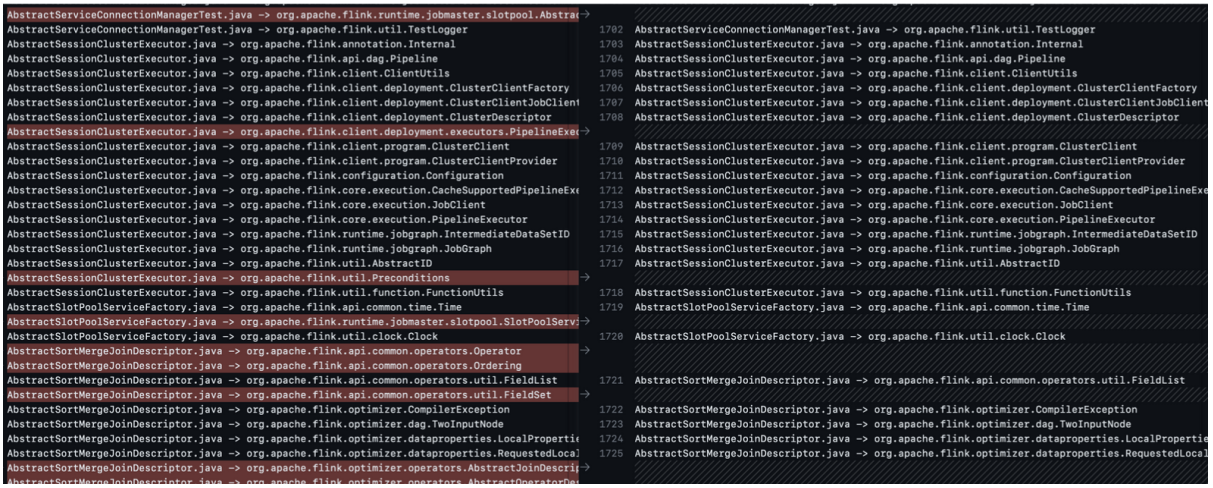

Figure 1 - Parsing Data from Understand

After parsing the data from Understand, we noticed certain dependencies that did not appear directly in the code. On closer observation, we noticed that it picked up classes that were not explicitly imported, but were in the package scope. For this reason, it included more dependency information than the other methods.

2 - Dependency Analysis Using Include

2.1 Include Tool

The initial phase of our dependency extraction process involves systematic traversal of the Flink source code directory structure. This thorough exploration aims to identify all Java files, discerned by the ".java" extension, through a methodical examination of directories and subdirectories.For each identified Java file, the script analyses the content to locate import statements. Specifically, it filters out the lines containing the keyword "import". However, to narrow down the focus to dependencies within the Flink architecture, the script specifically looks for lines that begin with "import org.apache.flink". This filtering ensures that only dependencies within the Flink ecosystem are considered. From the filtered import statements, the script extracts the relevant information needed for dependency analysis. For each line, it identifies the class (denoted as C) in the current Java file and the class it depends on (denoted as D). The format for each dependency is then represented as "C D". This information is crucial for establishing the relationship between different parts of the Flink architecture.

The extracted dependencies, in the specified format, are then stored in a TXT file. This file serves as the output, preserving the names of the files and the classes they depend on. This step ensures that the information is organised and can be easily referenced for further analysis or documentation.

2.2 Sequence Diagram

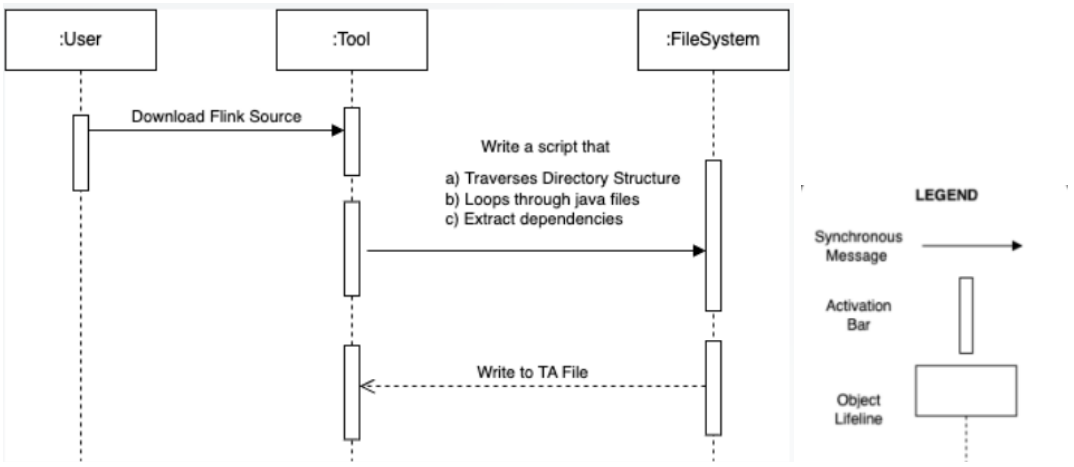

Figure 2 - Sequence Diagram Showing the Steps Involved in the Process

The sequence diagram illustrates a process where a user downloads the Flink source code, uses a tool to write a script that processes the code, and then outputs the results to a file:

1. The user initiates the download of the Flink source code.
2. A script is written to navigate the code's directory structure, process Java files, and extract dependencies.
3. The output is written to a TA file in the file system.

2.3 Observations from Include

Stored in the TA format, the exported files help in the comparison of the dependencies found by the include script to the other methods that are discussed in this report. Initially, for sake of simplicity and easier visualisation of dependency differences between methods, the files were extracted to a TXT file with a simple key-value mapping, with each line representing a dependency relationship between a file and one of its dependencies e.g. **"AIMDScalingStrategy.java -> org.apache.flink.annotation.PublicEvolving".** After sorting this file (as well as the output of other methods alphabetically, we were able to generate simple git diffs like the one in the figure below:

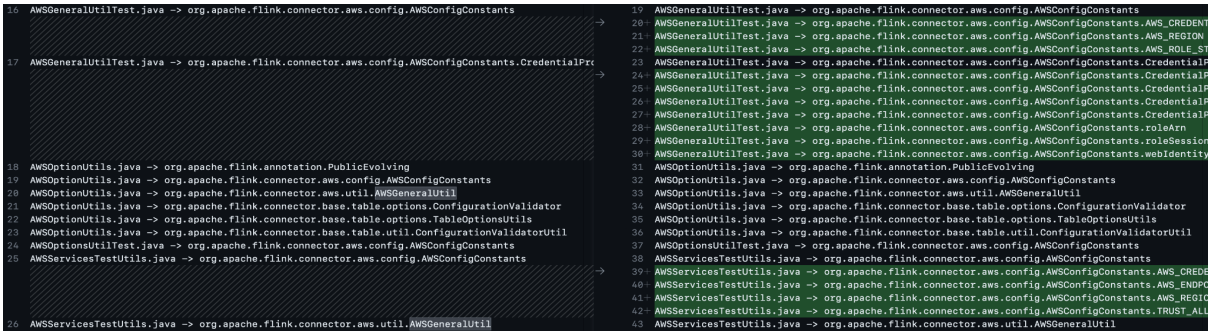

Figure 3 - Excerpt of Git diff between include dependencies and srcML dependencies

Doing this helped us to get a good overview of the differences between the dependencies that were picked up by alternative methods.

2.4 Advantages and Disadvantages

In our limited experience, the major advantage of the include method is its inherent simplicity. Our implementation only took ~43 lines of code, and was able to traverse the entirety of the Flink codebase and efficiently identify a significant number of explicit dependencies in the code. These dependencies help give a good overview of the overall system architecture and cover the major surface area of the system. With little effort, it can also be upgraded and tweaked to observe certain nuances in the code, making it more effective in finding more obscure dependencies.

While this method is able to do all of the aforementioned, its over simplicity is also a bane to its applicability in more comprehensive scenarios. Its inability to identify dependencies that are not explicitly imported (i.e. dynamic dependencies) makes it incapable of functioning in cases where absolute or near-absolute precision is required, and this is often the case in large-scale software re-architecture. As earlier mentioned, it can be improved to achieve this purpose, but only with significant and painstaking efforts to identify these "hidden" dependencies, in which case, another of the approaches described might be a better option.

3- Dependency analysis using srcML

3.1 srcML Tool

Our implementation of dependency extraction and analysis with srcML was a slightly more involved and convoluted process, compared to the include method. After running the srcML generation script, we considered methods of parsing it directly to generate a graph of the imports contained in it. Due to our unfamiliarity with using GPT to process the source code, we decided to parse the information and look at the imports contained within, to verify whether or not it contained more detailed information than the include method described above. Doing this successfully involved the conversion of the srcML XML output file to a JSON file for compatibility, as our scripts are predominantly written in JS (Node.js).

After looping through the imports that were found in this file, we were able to generate a similar key-value mapping to the one discussed in the include method above. The image below shows an excerpt of the output from this process:

```
AIMDScalingStrategy.java -> org.apache.flink.annotation.PublicEvolving
AIMDScalingStrategy.java -> org.apache.flink.util.Preconditions
AWSAsyncSinkUtil.java -> org.apache.flink.annotation.Internal
AWSAsyncSinkUtil.java -> org.apache.flink.annotation.VisibleForTesting
AWSAsyncSinkUtil.java -> org.apache.flink.connector.aws.config.AWSConfigConstants
AWSAsyncSinkUtil.java -> org.apache.flink.runtime.util.EnvironmentInformation
AWSAsyncSinkUtilTest.java -> org.apache.flink.connector.aws.config.AWSConfigConstants.AWS_ENDPOINT
AWSAsyncSinkUtilTest.java -> org.apache.flink.connector.aws.config.AWSConfigConstants.AWS_REGION
AWSAsyncSinkUtilTest.java -> org.apache.flink.connector.aws.util.AWSAsyncSinkUtil.formatFlinkUserAgentPrefix
AWSAuthenticationException.java -> org.apache.flink.annotation.Internal
AWSConfigConstants.java -> org.apache.flink.annotation.PublicEvolving
AWSCredentialFatalExceptionClassifiers.java -> org.apache.flink.annotation.Internal
AWSCredentialFatalExceptionClassifiers.java -> org.apache.flink.connector.base.sink.throwable.FatalExceptionClassifier
AWSGeneralUtil.java -> org.apache.flink.annotation.Internal
AWSGeneralUtil.java -> org.apache.flink.annotation.VisibleForTesting
AWSGeneralUtil.java -> org.apache.flink.connector.aws.config.AWSConfigConstants
AWSGeneralUtil.java -> org.apache.flink.connector.aws.config.AWSConfigConstants.CredentialProvider
AWSGeneralUtil.java -> org.apache.flink.util.ExceptionUtils
AWSGeneralUtilTest.java -> org.apache.flink.connector.aws.config.AWSConfigConstants
AWSGeneralUtilTest.java -> org.apache.flink.connector.aws.config.AWSConfigConstants.AWS_CREDENTIALS_PROVIDER
```

Figure 4 - Except from srcML extracted dependencies

3.2 Sequence Diagram

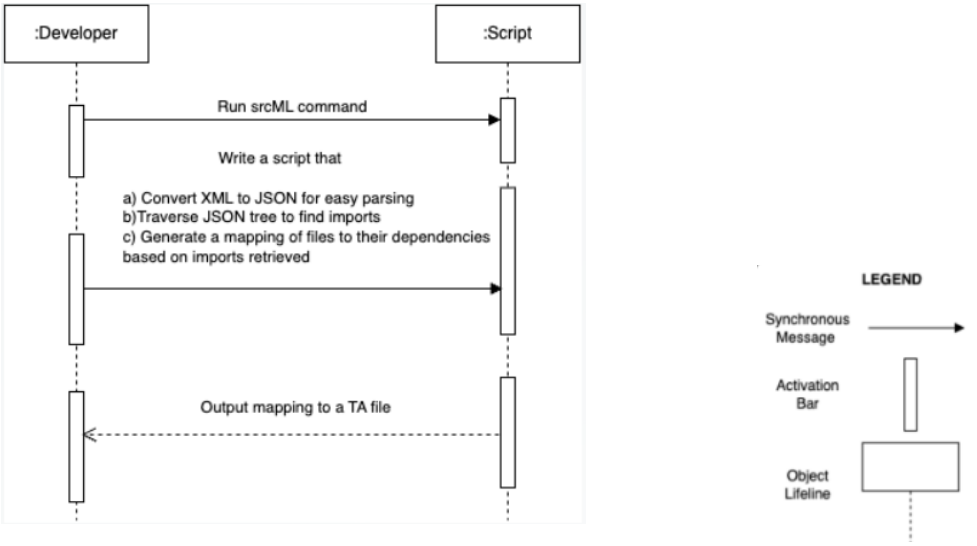

Figure 5 - Sequence Diagram Showing the Steps Involved in the Process

This sequence diagram shows a developer executing a script that:

1. Runs a srcML command to convert source code into an XML format.
2. The script then converts XML to JSON, searches the JSON for import statements, and generates a file-to-dependency mapping.
3. Finally, the script outputs this mapping to a TA file.

3.3 Observations using srcML

Because of the similarities in the methodology employed for the execution of the include and srcML methods, we were initially surprised at the seemingly divergent number of dependencies found by either approach. After employing our simple Git diff tool, we found that there were a number of dependencies that were found by the srcML method that were missing in our include implementation. After cross referencing with the source code, we discovered a running pattern. Our include implementation had missed all static imports, because of a bug in our match condition.As earlier stated, the files outputted were done in a key-value format similar to the include method for a greatly simplified comparison process with Git diffs. These were stored in a TXT format. The figure below shows the affected portion of code, and the fix (commented out):

```
for (const line of lines) {
  if (
    line.startsWith("import org.apache.flink")
    // || line.startsWith("import static org.apache.flink")
  ) {
    const split = line.split(" ");
    const last = split[split.length - 1];
    o += `${fn} -> ${last.replace(";", "")}\n`;
  }
}
```

Figure 6 - Excerpt of include script showing bug that was found because of srcML implementation

We observed that after including the required fix, both methods worked equivalently as expected.

3.4 Advantages and Disadvantages

When creating a script it will be able to scan imputed files for specific keywords. The script not only counts the occurrences of each keyword but also measures customizable metrics which can be used to specific needs of the analysis. The results are outputted as an XML format as it provides machine-readable structure with human-readable tags. Outputting as an XML file enabling further parsing for enhanced analysis of the keyword data.

There are a few constraints when utilising this tool, as it is unable to identify calls from X.java to the input file which can lead to missing important connections in the code. Also, when XML files that are generated may become cluttered with excessive tags. As this can impact the ease of reading and analysis. Lastly, it limits the language support for C, C#, C++, Java only.

4 - Quantitative Results

For the quantitative analysis, we used the generated ".txt" files , showing the dependencies found using the following approaches: include, srcML, and Understand. To understand the difference behind these approaches and to know which one is better, we used python and read the files as a dataframe. Then, we calculated the length of the dataframe to know about the number of values/dependencies found using each approach.

```
data1 = pd.read_csv("srcML-sorted.txt")
data2 = pd.read_csv("merged.txt")
data3 = pd.read_csv("include-sorted_main.txt")
data4 = pd.read_csv("understand-sorted.txt")


print("The Extracted Dependecies from the srcML approach are", len(data1))

print("The Extracted Dependecies from the include approach are", len(data3))

print("The combined dependencies from Understand, include, and srcML approach are:", len(data2))

print("The extracted dependencies from the Understand approach are:", len(data4))



The Extracted Dependecies from the srcML approach are 100791
The Extracted Dependecies from the include approach are 92792
The combined dependencies from Understand, include, and srcML approach are: 133991
The extracted dependencies from the Understand approach are: 118159
```

Figure 7 - Snippet Showing Extracted Dependencies from the Tools

After having the basic statistics about it, we then found the common dependecies between each approach to know how many dependencies were common and how many were exclusive to each approach. During the initial phase of identifying common dependencies, we encountered a challenge while employing the pandas dataframe implementation. Utilizing the "inner join" method posed an issue with handling "duplicate" entries within the dataset, and we sought to retain these entries without removing them. Consequently, we opted to exclusively utilize the text files to compute common dependencies within the dataset, avoiding the complications associated with duplicate entries. An illustrative example of this challenge is depicted in the image below.

```
with open('include-sorted_main.txt', 'r') as file1:
    content1 = {line.strip() for line in file1}

# Read content from the second text file and store lines in a set
with open('srcML-sorted.txt', 'r') as file2:
    content2 = {line.strip() for line in file2}


with open('merged.txt', 'r') as file3:
    content3 = {line.strip() for line in file3}

with open('understand-sorted.txt', 'r') as file4:
    content4 = {line.strip() for line in file4}

# Doing the analysis between merged and srcML approach
# Find common lines using set intersection
common_lines = content2.intersection(content3)
print("The common elements out of the Merged and srcML approach are:",len(common_lines))
true_positive= len(common_lines)

The common elements out of the Merged and srcML approach are: 100178
```

Figure 8 - Snippet Showing Common Elements in Two Tools

5 - Qualitative Results

A qualitative analysis was used to help us better break down, understand, and visualise the three dependency extraction techniques we used to get an inside look at the architecture of Apache Flink. This helped us understand that software architecture analysis is not a straightforward endeavour and can be done in several ways which can give different results each time, depending on how you decide to set up the extraction analysis, and that some techniques are more effective than others. Our qualitative results are from the three distinct methods of software dependency extractions we used in our findings, Include, srcML, and Understand. The comparison helped us see the pros and cons of each of the methods used which gave us an insight into when a technique is best employed, and the limitations that come from using a specific method. To identify these, we calculated the Precision and Recall of each of the tested methods when comparing their results to each other, as well as created a Venn Diagram to visualise the results of the comparisons between the different dependencies extraction methods.

To test the comparative efficacy of each method, we first calculated the appropriate sample size to use for the analysis. We calculated the size of our population to be 133993, by merging the outputs of each method and removing duplicate lines. With a chosen confidence level of 95% and a confidence interval of 5, this yielded a sample size of 383 random dependencies.

After ~15 iterations, we noticed that the include and srcML methods matched 68.8% and 68.9% of the dependencies in the sample on average, while the Understand method matched 99.98% of the dependencies. The close correlation of the outcomes for the include and srcML method apparently stem from their inherent methodological similarities. These numbers also reinforce the fact that the Understand method is the most accurate one of the three.

5.1 - Precision and Recall

A) Now, we wanted to calculate the precision and recall of each employed method to understand the underlying differences between them, and to see which one is performing better. We took the common approach of calculating True positive, False Positive, and False negative to calculate the precision and recall, along with the percentage of common samples with each approach. For all of these comparisons with the approaches, we compared dependencies found with each method to the total number of combined dependencies. The screenshot and the results for each approach are shown in the photos below. The **Precision** was calculated with the following formula in accordance with our methodology: **(True Positive)/ (True Positive + False Positive)**

Meanwhile, the **Recall** was calculated with the following formula in accordance with out methodology: **(True Positive)/(True Positive + False Negative)**

```
print("The false negative is:",len(data1) - len(common_lines) )
false_negative = len(data1) - len(common_lines)

print("The False positive is:", len(data2)- len(common_lines))
false_positive = len(data2)- len(common_lines)
```

```
The false negative is: 613
The False positive is: 33813
```

```
print("The Precision for this comparison:", true_positive/(true_positive+ false_positive) )
print("The Recall is", true_positive/(true_positive + false_negative ))
print("Percentage of common samples with the srcML approach", (true_positive/len(data2))*100)
```

```
The Precision for this comparison: 0.7476472300378384
The Recall is 0.9939181077675586
Percentage of common samples with the srcML approach 74.76472300378384
```

Figure 9 - Snippet Showing Recall and Precision Calculations Using Approach 1

```
# Doing the analysis between merged and include approach
common_lines = content1.intersection(content3)
print("The common elements out of the Merged and include approach are:",len(common_lines))
true_positive= len(common_lines)

print("The false negative is:",len(data3) - len(common_lines) )
false_negative = len(data1) - len(common_lines)

print("The False positive is:", len(data2)- len(common_lines))
false_positive = len(data2)- len(common_lines)


print("The Precision for this comparison:", true_positive/(true_positive+ false_positive) )
print("The Recall is", true_positive/(true_positive + false_negative ))
print("Percentage of common samples with the include approach", (true_positive/len(data2))*100)
```

```
The common elements out of the Merged and include approach are: 92212
The false negative is: 580
The False positive is: 41779
The Precision for this comparison: 0.6881954758155399
The Recall is 0.9148832733081327
Percentage of common samples with the include approach 68.81954758155399
```

```
## Understand Approach
common_lines = content4.intersection(content3)
print("The common elements out of the Merged and include approach are:",len(common_lines))
true_positive= len(common_lines)

print("The false negative is:",len(data4) - len(common_lines) )
false_negative = len(data4) - len(common_lines)

print("The False positive is:", len(data2)- len(common_lines))
false_positive = len(data2)- len(common_lines)


print("The Precision for this comparison:", true_positive/(true_positive+ false_positive) )
print("The Recall is", true_positive/(true_positive + false_negative ))
print("Percentage of common samples with the understand approach", (true_positive/len(data2))*100)
```

```
The common elements out of the Merged and include approach are: 117513
The false negative is: 646
The False positive is: 16478
The Precision for this comparison: 0.8770215910023808
The Recall is 0.99453279056187
Percentage of common samples with the understand approach 87.70215910023808
```

Figure 10 - Snippet Showing False Negative/Positive and Precision/Recall Numbers Using Approach 1

B) Now, we wanted to calculate the precision and recall of each employed method to understand the underlying differences between them, and to see which one is performing better. The approach we took to calculating the Precision and Recall was to compare the number of unique dependencies extracted from each of the methods, to the total number of dependencies found by the methods, which we will call Precision, as well as compare them to the total number of dependencies found by all the methods together, which we will call Recall.

Precision = unique/found x 100%

Recall = unique/total x 100%

Include:

92794 dependencies found

582 unique dependencies

133993 total dependencies

Precision = 582/92794 x 100% = 0.6271957

Recall = 582/133993 x 100% = 0.4343510

srcML:

100792 dependencies found

8580 unique dependencies

133993 total dependencies

Precision = 8580/100792 x 100% = 8.51258

Recall = 8580/133993 x 100% = 6.40332

Understand:

118160 dependencies found

34461 unique dependencies

133993 total dependencies

Precision = 34461/118160 x 100% = 29.16469

Recall = 34461/133993 X 100% = 25.71851

5.2 VENN Diagram

The following Venn Diagram highlights the unique and shared dependencies between the Include, Understand and srcML extraction methods. In all, there are a combined 83,699 dependencies extracted between the three met. There were 8513 dependencies extracted that were common between scrML and Include. And there were no common extracted dependencies between Understand and Include, and between Understand and srcML that was uniquely common between the two methods, always being seen in all three extractions.

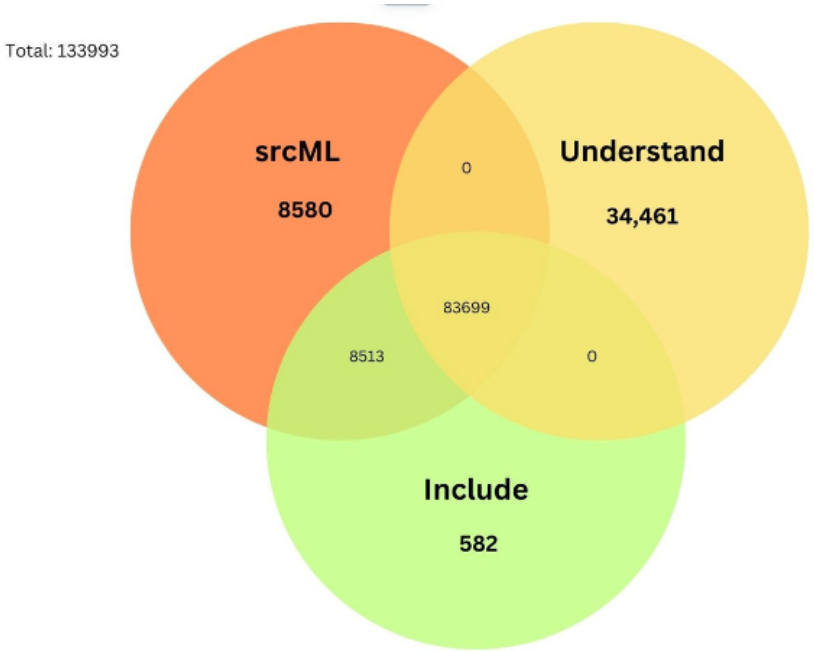

Figure 11 - Unique & Shared Dependencies Between Three all Three Tools

Limitations

The study is subject to limitations, primarily stemming from the random nature of the sample selection, raising concerns about its representativeness for the entire system under examination. Furthermore, the techniques employed may encounter challenges in accurately recognizing specific dependency types, particularly those intertwined with intricate language features. Language-specific idiosyncrasies introduce variability, impacting the types of dependencies identified and extracted by each technique. The analysis also reveals the presence of false positives and negatives within the extracted dependencies, casting a shadow on the reliability of the results. Additionally, the occurrence of duplicated dependencies across multiple methods introduces a precision-related challenge in our calculations, further contributing to the nuanced limitations inherent in the employed techniques.

Potential Risks

The utilisation of extraction tools in our project introduces potential risks that warrant consideration. In the case of the “include” method, the manual maintenance and updating of inclusion patterns may pose challenges and become susceptible to errors as the project evolves over time. This introduces the risk of inconsistencies and inaccuracies in the extracted dependencies, necessitating vigilant oversight. For the last dependency extraction tool we used, the accuracy of dependency extraction is intricately tied to the correct parsing of the source code. Any deviation or inaccuracy in parsing may lead to erroneous results, compromising the reliability of the extracted dependencies. Additionally, there is a potential risk that certain dependencies may not be handled correctly, introducing a source of uncertainty and emphasising the importance of thorough validation and verification processes in our dependency analysis. These risks underscore the need for a cautious and vigilant approach when relying on extraction tools, emphasising the importance of continuous monitoring and adaptation to evolving project dynamics.

Lessons Learned

Throughout this project, we have gained valuable insights and lessons that contribute to our understanding of dependency extraction in software systems. Exploring various techniques has broadened our knowledge on the diverse methods available for dependency extraction. We recognize the significance of carefully selecting and evaluating the size and representativeness of the sample portion for qualitative analysis, as it plays a pivotal role in drawing accurate conclusions about the entire system. An essential lesson emerged regarding the distinct characteristics inherent to different programming languages, influencing the nuances of dependency extraction. Our exploration involved learning the intricacies of three distinct tools for dependency extraction, highlighting the importance of tool selection in the analytical process. Furthermore, we gained proficiency in calculating precision and recall, emphasising their role in assessing the reliability of our analysis. A crucial takeaway is the acknowledgment of the necessity to employ multiple tools, as this approach ensures a more comprehensive and robust coverage of dependency extraction, allowing for a more thorough and accurate analysis of software systems.

Conclusion

Conclusively, our exploration of diverse dependency extraction methodologies for software systems has yielded significant revelations and teachings that augment software development methodologies. We now have a thorough grasp of the advantages and disadvantages of each strategy because of the investigation of tools like Understand, writing a script that searches for “include” directives and srcML. Throughout our work, it has become clear how important it is to carefully choose your samples for qualitative analysis, how programming language features affect dependency extraction, and how important precision and recall calculations are. Acknowledging the significance of using various tools, we stress the necessity of a comprehensive strategy for dependency extraction in order to guarantee a comprehensive and

precise examination of software systems. We have concluded that the Understand tool was the best tool in analysing the dependencies. It provided us with comprehensive results.

The following are our proposals for future directions:

1. Promote joint efforts among developers to mould the direction of dependency extraction tools in the future. Through collaborative effort, these tools can develop with a range of viewpoints, strong features, and increased language support to meet the changing demands of a large developer community.
2. It is important to establish thorough documentation procedures, the reasoning behind design choices, and the lessons discovered from past endeavours.

The following is a summary of our key findings:

1. **Various Dependency Extraction Methods:** Our research identified a number of different dependency extraction methods, each with advantages and disadvantages.
2. **Significance of Sample Size:** In qualitative analysis, the size and representativeness of the sample fraction were shown to be important.
3. **Importance of Tool Selection:** The significance of tool selection in dependency extraction was highlighted by our investigation of two distinct tools. The precision and thoroughness of the retrieved dependencies are directly impacted by the toolset selection.