Apache Flink

Dependency Extraction

Group Small:

Muhammad Omar

Ruth Bezabeh

Nishiket Singh

Rathul Momen

Corneille Bobda

Bartolomeo Gisone

Abstract

This paper presents a comprehensive exploration of dependency extraction techniques in Apache Flink, a popular distributed stream processing framework. Two primary techniques, "Understand" and "Import Statements," are examined in detail. "Understand" utilizes code analysis to extract dependencies between entities at the source code level, emphasizing file-to-file, class-to-class, and architecture-to-architecture relationships. In contrast, "Import Statements" focuses on extracting dependencies through import declarations. The study identifies and addresses issues encountered during implementation, including challenges with srcML, JDT, and JavaParser. The theoretical foundation involves tracing method invocations, identifying new keywords, matching method definitions, handling external dependencies, and including transitive dependencies. The paper concludes with a comparison process involving quantitative and qualitative analyses, highlighting the rationale for observed differences and emphasizing the risks, limitations, lessons learned, and areas for improvement in the applied techniques.

Introduction

Dependency analysis is a crucial aspect of understanding and managing complex software systems, and Apache Flink, as a distributed stream processing framework, poses unique challenges in this regard. This paper delves into two distinct techniques for dependency extraction: "Understand" and "Import Statements." The former employs code-centric analysis, offering a nuanced view of dependencies at various levels, while the latter relies on import declarations for extraction. The study explores inherent issues encountered during the implementation, particularly related to srcML, JDT, and JavaParser. The theoretical underpinning involves tracing method invocations and addressing challenges related to handling external dependencies and transitive relationships. A thorough comparison process is undertaken, incorporating both quantitative and qualitative analyses to provide a comprehensive understanding of the strengths, weaknesses, and nuances of each technique. The paper concludes by reflecting on lessons learned, potential areas for improvement, and the importance of continuous learning in refining dependency extraction methodologies in the context of Apache Flink.

Technique 1 - Understand

[1] Understand is a tool that can be used to extract dependencies at the source code level. Entities in the project are first grouped into files, architecture, e.g. /src, or more, based on the directory structure. It can detect file to file, class to class and architecture to architecture. The dependency extraction follows these steps.

1. Find all entities in the starting group.
2. Get the references for each entity
3. Determine the group at the given level for the second (referenced) entity
4. If the group of the second entity exists and is not the same as the starting group, then the reference is a dependency connecting the two groups.

[2] The extracted dependency is then saved in a csv which is then converted into a TA [3] formatted file using perl. That can then be used to analyze and visualize the dependencies, and is what was used to compare the dependencies between the different techniques.

• Technique 2 - Import Statements

For technique two, we used scripts to extract dependencies from import statements. we ran the 1st script that traverses through the project directory to find the files. it then extracts and saves the imports in that file as well as the rest of them.

```python
import os
import re

def extractDependencies(flink):
    flinkDependencies = {}

    pattern = re.compile(r'^\s*import\s+org\.apache\.([\w.]+)\s*;')

    for root, _, files in os.walk(flink):
        for file in files:
            if file.endswith('.java'):
                filePath = os.path.join(root, file)
                relativePath = filePath.replace(flink, '').lstrip(os.path.sep)
                with open(filePath, 'r', encoding='utf-8') as java_file:
                    dependencies = []
                    lines = java_file.readlines()
                    for line in lines:
                        match = pattern.match(line)
                        if match:
                            dependency = match.group(1)
                            dependencies.append(dependency)
                    flinkDependencies[relativePath] = dependencies

    return flinkDependencies
```

Fig 1.0 script for extracting dependencies from import

We then ran a second script that parses through the first output file to convert it to a TA format that will make it easier for our analysis process. A third script is ran to re-organize the ta-formated file so it can be the same structure as the output from the understand extraction.

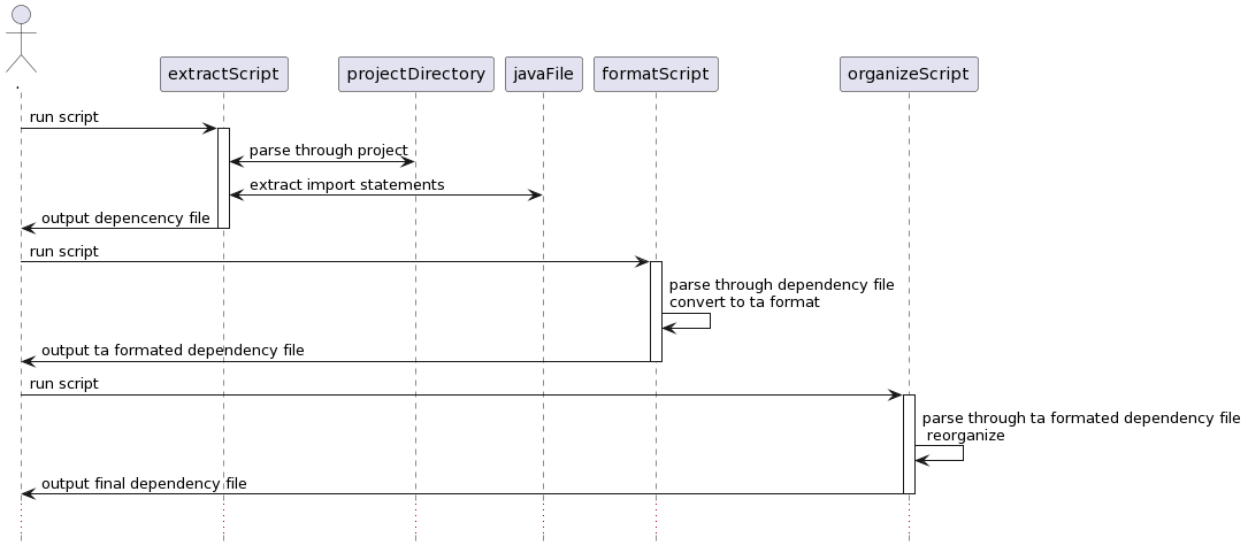

Fig 1.1 process taken for extracting dependencies from import

- # Technique 3 - Multiple

After continuous attempts we were not successful in completing the dependency extraction due to some issues.

1. srcML Issues:

Could Not Read XML: There were issues with reading the generated XML, it was due to XML parsing errors or corruption during the conversion process.

Fig 1.2 srcml parsing errors

2. JDT Issues:

IDE does not recognize JDT as a dependency. JDT installation via Eclipse Marketplace raises concerns as it did not work for anyone in the group. Manually installed JDT requires confirmation that the JAR files or dependencies are correctly included in the project, which upon further examination was found to be an IDE error. AST parser from JDT could not be resolved, possibly due to missing JAR files or incorrect configuration.

3. JavaParser Issues:

Maven cannot find the JavaParser dependency; inspect the project's pom.xml file for typos or errors in the dependency declaration and confirm the correct configuration of Maven repositories.

Despite of the issues here is what we would have done if we had been successful.

- Trace Method Invocations: Start by identifying method invocations in your codebase. This can be done through static or dynamic analysis, depending on whether we're looking at the code without executing it (static) or observing its behavior during runtime (dynamic).

- Identify New Keywords: Look for new keywords introduced in the code that signify the initiation of a method call. This would include language-specific keywords like call or invoke. If using a dynamic analysis approach, we may need to instrument your code to capture method invocations during runtime.

- Matching Method Definitions: Once we identify a method invocation, trace it back to its corresponding method definition. This involves locating the source file or class where the method is defined.

- Handle External Dependencies: Consider how to handle external dependencies such as libraries or frameworks. If a method is defined in an external library, we wouldl need to account for this in your dependency analysis.

- Include Transitive Dependencies: Consider whether we want to include transitive dependencies. Transitive dependencies are dependencies of dependencies. For example, if method A calls method B, and method B calls method C, then method A has a transitive dependency on method C.

● Comparison Process

- We format the include dependency file to match TA format
- We wrote a script to compare the two methods
- The discrepancies were manually analyzed

```
def understandCount(filePath):
def includeCount(filePath):
def understandInstances(understandFile):
def includeInstances(includeFile):
def commonFiles(understandFile, includeFile):
def commonDependencies(understandFile, includeFile):
def extractSmaples(understandFile, includeFile):

understandFile = 'UnderstandDependencyFile.raw.ta.txt'
includeFile = 'reorganized_ta_format.txt'
```

Fig 1.3 comparison process

• Quantitative Analysis

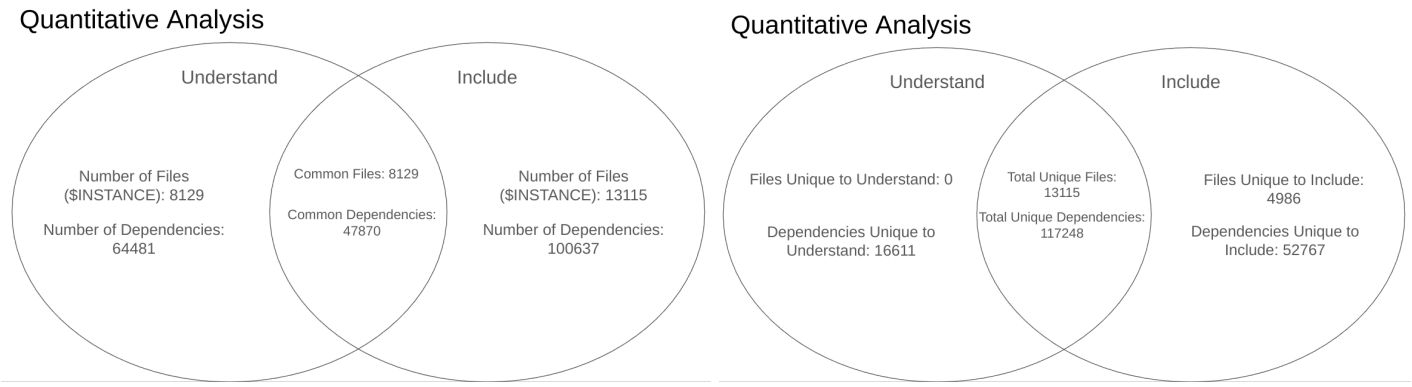

Fig 1.4 Quantitative analysis

In terms of our quantitative analysis with Understand and Include, the Understand tool 64481 dependencies. This count represents the dependencies identified through the Understand tool, which focuses on code analysis to extract various types of dependencies. It suggests that Understand identified a substantial number of relationships within the codebase. The Include tool found 100637 dependencies. The higher count from the Include extraction indicates a broader measure of dependencies, encompassing not only code-level interactions but also all files included in the project, such as headers, test files, or other non-essential components. In terms of the number of files, the Understand extraction found 6129 files and the Include extraction found 13115 files. The number of files

identified by Understand suggests the subset of files actively contributing to the dependencies detected. This count is likely more focused on essential code components. The higher count from the Include extraction includes all files in the project, providing a more comprehensive view that encompasses not only essential code but also potential non-essential files, tests, headers and more. Using a script to detect common dependencies, it was found that both the Understand and Include extractions had 47870 shared dependencies and 8129 common files. The fact that 47,870 dependencies are common between Understand and Include extractions indicates a substantial overlap. The 8,129 common files further reinforce that a significant portion of the files contributing to dependencies is identified by both extraction methods.

The extraction found 16611 dependencies unique to Understand and 52767 dependencies unique to Include. It also found 0 unique files to Understand and 4986 unique files to Include.

- # Qualitative Analysis
-

The qualitative analysis portion was done with a variety of methods. A stratified sampling method was used to find the sample size used to help calculate the precision and recall used for the qualitative analysis. The confidence level used was 95%, the confidence interval used was 5 and the total population was 117248. This population is obviously the number of dependencies found within the App. After using the stratified sampling method, a sample size of 383 was calculated. Furthermore, there were 16611 dependencies exclusively found by the Understand method, 52767 dependencies found by the Include Method, and 47870 dependencies that were found by both of them.

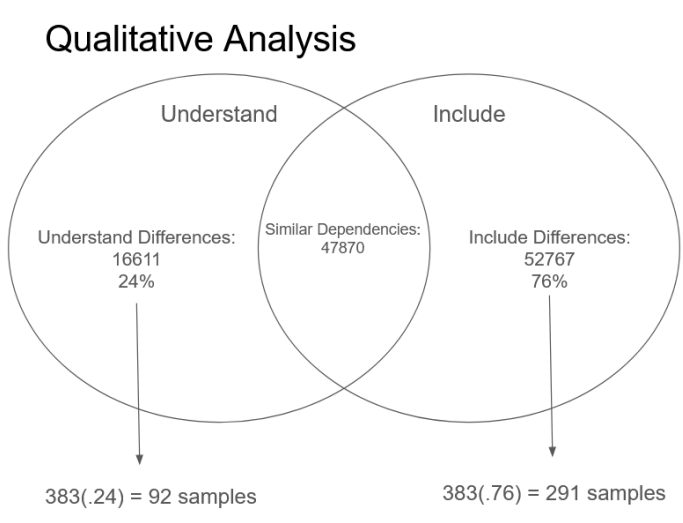

Fig 1.5 Qualitative analysis

In order to calculate the precision and recall between the two subjects, a sample of dependencies were chosen at random, up to 383 in order to calculate them. When done so, we gathered a sample size of about 4 common dependencies, 88 dependencies found that were unique to the Understand method, and 287 dependencies

that were unique to the Include method. As a result, we ended up having very skewed numbers for our calculations.

Precision and Recall For Understand:

Precision: 5/(88+5)= 0.052=5%

Recall: (5/(5+287))=0.017=1.72%

Precision and Recall For Include:

Precision: 5/(287+5)= 0.052=5%

Recall: 5/(5+88):=0.017=1.72%

The results are obviously very poor, but unfortunately, that's the potential risk you run by gathering a random sample. Because the amount of common dependencies were so low, it drastically brought the precision and recall for both analysis methods as a result.

- # Rationale for Differences

  - 1."Understand" analyze codebases to extract various metrics, including dependencies between different components (files, classes, functions, etc.).
  - 2. Understand can identify dependencies based on function calls, variable references, and other interactions between different parts of the code.
  - 3. The number of "Understand Files" (8129) likely represents the number of source files being analyzed, and "Include Files" (13115) might indicate the total number of files, including headers or other included files.
  - 4.The number of "Include Files" (13115) suggests the total number of files that are included in the project. Including files doesn't necessarily mean those files contribute to the dependency graph. Some included files might be header files, test files, or other non-essential components. The inclusion of test files could explain why there are more files in the "Include" count than in the "Understand" count.

Fig 1.6 list of files in the understand file

- 5."Understand" tools focus on actual code interactions and dependencies, providing a more refined view of the true dependencies in the codebase. "Include" statements, on the other hand, are a broader measure, encompassing all files that are part of the project, including headers and potentially non-essential files like tests.

- 6.In summary, the differences between dependency extraction using "Understand" and counting "Include" statements could be due to the comprehensive nature of the latter, which includes all files, including tests.

● Risks and Limitations

A risk when using Understand is that it may not find all the dependencies. As previously mentioned, using Understand without using ‘include’ statements, leads to a loss of number of files being analyzed.

Due to the nature of the different techniques, finding dependencies is limited to the individual technique. Because of this, there is no single technique which is significantly less effective than another, as all techniques contain some dependencies not found in others. This means that even though there is overlap from the results, using all three techniques provides the most robust list of dependencies.

Due to difficulties with srcML, completing a portion of the assignment proved difficult. Because of the unfamiliarity with srcML, there are limitations in the information we could gather.

● Lessons Learned

Things We Regret or Would Do Differently:

- Familiarize with srcML:

If there are regrets or areas for improvement, it may involve not dedicating enough time to familiarize ourselves with srcML. This includes understanding its features, capabilities, and the other tools offered by srcML including srcSlice.

- Better Understanding of Understand:

Similar to srcML, regrets may stem from not having a thorough understanding of the capabilities and functionalities of tools like Understand. This includes its analysis methods, configuration options, and how it handles specific code patterns.

In summary, awareness of the nuances in different techniques, the benefits of employing multiple methods, and a deeper understanding of specific tools can significantly enhance the accuracy and completeness of dependency extraction in a codebase. Continuous learning and adaptation based on feedback and experiences contribute to more effective and reliable dependency analysis.

References

[1]https://support.scitools.com/support/solutions/articles/70000583144-how-dependencies-in-understand-are-determined
[2]http://www.cse.yorku.ca/~zmjiang/teaching/eecs4314/slides/EECS4314ArchRecoveryLabNotes.pdf
[3]https://plg.uwaterloo.ca/~holt/papers/ta-intro.html