EECS 4314

Assignment 4

Dependency Extraction Report - Apache Flink (v1.17.1)

December 4, 2023

The Debeggars

Alain Ballen - alain612@my.yorku.ca
Arjun Kaura - arjunka@my.yorku.ca
Davyd Zinkiv - dzinkiv@my.yorku.ca
Nilushanth Thiruchelvam - nilut@my.yorku.ca
Peter Lewyckyj - peterlew@my.yorku.ca
Xu Nan Shen - jshenxn@my.yorku.ca

Abstract

This report entails an analysis on dependency extraction methods on the Apache Flink software. We used three different approaches for extracting dependencies including utilizing the Understand software, using jdeps to show class-level dependencies through import statements, as well as srcML to convert the source code to XML and perform further analysis on. Each tool has its own set of strengths and weaknesses, which are discussed within the report. The extraction process is described for each strategy and a quantitative analysis is performed on the results including calculating the precision and recall for each tool. Finally, the limitations of each strategy is discussed with its implications on the results obtained through the analysis.

1. Alternatives Considered

1.1. POM-Parsings

POM Parsing can read any pom.xml and all the xml element attributes are merged into a parent element. Both the xml string and the parsed object are returned. Parsing options can be provided by the user. The issue with this method is that because it only utilizes the pom files, the parsing only returns the very-top level (package level) dependencies and does not reveal any subsystems and classes. For instance, all the Flink run-time cluster subsystems (such as JobMaster, TaskManagers etc.) are grouped under Flink-runtime. This is not a sufficient level of dependency detail for the purpose of this study.

1.2. Jarviz (ASM opcode analysis)

Most software modules have many interdependencies, potentially leading to low-cohesion and high-coupling among software modules. Jarviz is a tool that can help to understand these intricate dependencies to a more understandable level.

Jarviz is a dependency analysis tool for Java applications. Non-private methods in Java classes can be accessed by other classes, and as a result, method calls can be intertwined creating manifold couplings among each other. Jarviz deeply analyzes Java bytecode to reveal these couplings in a user-friendly format. It scans binary artifacts using a custom classloader and generates a complete graph of dependency couplings between methods across multiple artifacts.

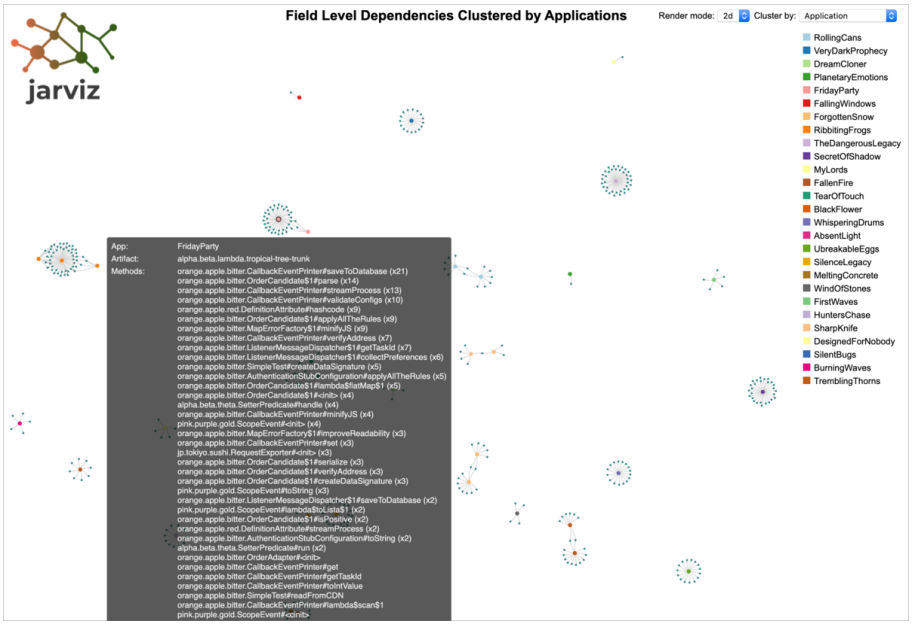

Figure 1: *Sample Jarviz dependency graph*

Jarviz consists of three modules:

- a Java library to scan dependencies in binary artifacts
- a Node.js module to generate a visual graph
- and a command-line-interface to run it easily in a shell

Jarviz Use Case:

Sample Coupling Data


```
{
 "appSetName": "FooPortfolio",
 "applicationName": "MyApp",
 "artifactFileName": "foo-product-1.2.1.jar",
 "artifactId": "foo-product",
 "artifactGroup": "com.foo.bar",
 "artifactVersion": "1.2.1",
 "sourceClass": "foo.bar.MyClass",
 "sourceMethod": "doFirstTask",
 "targetClass": "foo.bar.YourClass",
 "targetMethod": "getProductId"
}
```


1.3 jdeps

Figure 2: *jdeps extraction process*

We initially utilized the jdeps tool, a command line tool that is able to extract class-level dependencies of Java class files from a JAR file. The jdeps command was run against the Flink-dist JAR which includes all the application related projects and their classes. A dependency DOT file mapping all the statically declared dependencies was produced and a PERL script is utilized to convert the file into TA format.
In later processes of qualitative/quantitative analysis done with jdeps, we found that due to several existing bugs of the tool, the tool was not able to extract accurate and a satisfactory number of dependencies. For instance, in the entities ($INSTANCE) extractions, there are entities with (not found). In addition, using inference techniques (explained in section 5), we discovered that the accuracy of dependencies was extremely low (70% false positives).
Please note that at the time of the presentation, this was used as our second method. However, as a result of the mentioned findings, this method was not used as part of our final studies and results.

2. First Method: Understand

2.1. Tool Overview

For our first approach, we used UnderstandTM software developed by SciTools. Amongst the many features it offers, it allows for the extraction of file dependencies. The tool was used in Assignment 2 for concrete architecture recovery.

In Understand, an entity is any element Understand has information on, such as files, classes, functions, or variables. References connect two entities, and dependencies are references connecting two groups of entities based on their parent-child relationships. The parent of an entity is determined during analysis, usually based on the "Define in" reference kind. Entities

can be grouped by architectures, and dependencies are calculated at different levels of the parent chain, such as file, class, and architecture. The calculation involves finding entities in the starting group, getting their references, determining the group for the referenced entity at a given level, and identifying dependencies when the groups are different. Optimizations are applied to prevent entities from belonging to multiple groups, depending on the level. References are filtered based on a reference kind string, and not all references are considered as potential dependencies [2]. For the purposes of this assignment, we assume the Understand parser's dependencies mapping to be a "black box" technique.

2.2. Extraction Process

Due to the nature of this method, the extraction process is quite simple when compared to other methods. Using UnderstandTM software, we extracted the source code file dependencies, in a CSV format. From there we used the transformUnderstand.pl script (provided in the lab materials) to convert it to raw.ta format.

Figure 3: *UnderstandTM extraction process*

3. Second Method: "Import" Parsing

3.1. Extraction Process

The Import Parsing method we utilized is implemented in C# (Source code: see Appendix, item 1). The implementation does a full recursive search in the Apache Flink source code for all java class codes. Then using Regex, and in every file, matches for import statements that include org.apache.flink. Consequently, all matches are written into the output TA file in TA format accordingly. This method is reliant on the consistent usage of "import" statements by the Apache Flink developers. There may be instances where there are redundant import statements (unused package specified), and/or missing import statements (explicit definition of the package path in the code without the use of "import").

4. Third Method: srcML

4.1. Tool Overview

srcML is a tool which converts source code to XML. It's compatible with C, C++, C#, and Java. The XML output can also be converted back to source code. This tool makes source code analysis easier because XML is a popular format which has support in most languages. srcML itself is also very easy to use as code can be converted with a single command without any specific configuration. Unfortunately srcML seems to be no longer actively maintained since the last commit was in 2021 and the prebuilt binaries available for download are all for older operating system versions. Finally, the XML output by itself isn't very helpful. Further analysis still needs to be performed to obtain any meaningful insights.

4.2. Extraction Process

To perform the extraction we need to download the srcML from an official website [3]. Because it's not actively maintained, some of the installation packages are not up to date, and require some workarounds to install the tool. To convert the source code to XML format we run the command `srcml --verbose flink-1.17.1-src.tgz -o flink.xml` which takes Flink zip file and outputs the .xml file. Having XML format now allows us to access specific pieces of the source code individually. We wrote a Python script that uses xml.etree.ElementTree [4] module to parse the xml tree to search each file for object instantiations, method calls, constructor and method arguments. Once parsed completely the script constructs the CSV file with two columns: "From File" and "To File", containing the dependencies. Then we used transformSrcML.pl script to convert data from the CSV file to produce raw.ta file. This script is a modified version of the original transformUnderstand.pl used in Assignment 2 but it considers only the first two columns of CSV.

Figure 4: *srcML extraction process*

5. Quantitative Analysis

5.1 Process

The TA files extracted based on the three methods, Understand, Import parsing, srcML parsing (Section 2 - 4) are inputted into a comparison program written in C# (Source code: Appendix item 2).

The program processes the individual TA files and maps the dependencies accordingly in 3 separate dictionaries. Quantitative counting results are then outputted into a QRESULT output file which showcases the differences between the 3 methods of extraction used. In order to facilitate further qualitative analysis, the program calculates the number of dependencies based on the following seven categories: dependencies extracted only by: Understand, Import parsing, srcML parsing, dependencies extracted by and only by: Understand and Import Parsing, Understand and srcML Parsing, Import Parsing and srcML Parsing, and dependencies that were extracted by all 3 methods.

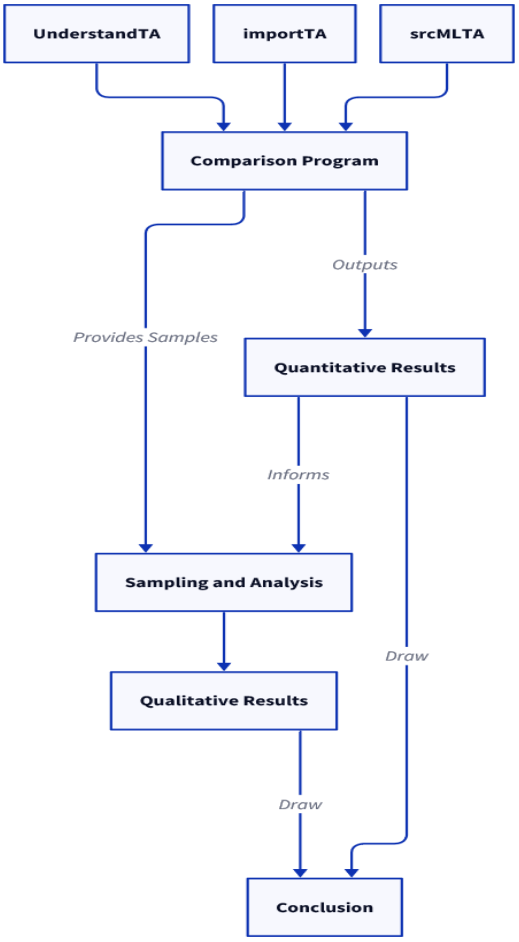

Figure 5: *Qualitative/Quantitative Analysis Process (Full-resolution link: Appendix item 3)*

5.2 Results

Across the 3 methods, a total of 135116 dependencies were extracted. Import extracted 92562 (68.5% of total) dependencies, srcML extracted 67387 (49.8% of total) dependencies, and Understand extracted 118793 (87.9% of total) dependencies. 48753 (36.1% of total) dependencies were extracted by all 3 methods. 4974 (3.7% of total) were extracted by srcML and Understand but not by Import. 3031 (2.2% of total) were extracted by Import and srcML but not by Understand, and 38115 (28.2% of total) were extracted by Understand and Import but not by srcML.

For the uniquely extracted dependencies, the program also produced an entity count based on where the dependencies are mapped from (the dependent class). Understand dependencies were mapped over 9425 entities (~3 dependencies per entity). srcML dependencies were mapped over 4300 entities (~2.5 dependencies per entity). Import dependencies were mapped over 1194 entities (~2.2 dependencies per entity).

Figure 6: *Venn Diagram showing intersection of dependencies and count*

Based on the above findings, we can conclude that the Understand method was able to extract the highest number of dependencies with a higher dependencies count per entity (in Understand case, file-based entities). The import method is second in the total number of extraction, but does not extract a high number of dependencies that are different from the ones extracted by Understand (only 2663 +3031 = 5694). srcML extracted less than half of the total number of dependencies extracted, but had a relatively high number of unique dependencies extracted that were not discovered by either Understand or Import, 10629 (7.9% of total). Based on our understanding of the methods, we can infer at this point that Understand is the most comprehensive dependency extraction method that covers the majority of what import statements establish in dependencies. The amount of srcML-unique dependencies extracted are likely caused by the difference in approach, as it is a class-based extraction rather than a file-based dependency extraction.

We then qualitatively analyze the accuracy of the 7 categories of dependencies extracted to draw further conclusions on the three methods.

6. Qualitative Analysis

6.1 Process

Based on the dependencies count produced in Section 5, we utilize the statistical inference model (Sample size calculator:https://www.surveysystem.com/sscalc.htm) to extract a sample size in order to analyze the validity of the dependency extractions. We used a Confidence Level of 95% with a Confidence Interval of 10 due to time constraints and limited human resources. The random sample is provided within the same comparison program that is used to produce the quantitative results (Source code: Appendix item 2). The list of samples are then manually analyzed by our team to assess whether an individual dependency is valid. Then by using the sample size validity percentage. We can calculate the precision and recall rates of the three methods that were utilized and draw further conclusions.

6.2 Results

Common Dependencies of all three methods:
Total size: 48753, Sample size: 96.
Number of invalid dependencies found: 0
Explanation: Within the sample that were examined, all dependencies that belong to this group are valid dependencies. There are obvious relations between the dependent entity and the depended entity, including direct function calls (both static ones and object instance calls), implementations of data types, singleton usage, abstracted interface implementations etc.
Validity rating: 100%

Dependencies only extracted by Understand:
Total size: 26951, Sample size: 96.
Number of invalid dependencies found: 4
Explanation: Within the sample that were examined, we found that unlike the other two methods, Understand is able to process dependencies related to python and other file types. For example, the pip_test_code python file has dependencies to shell.py, schema.py, expressions.py and table_descriptor.py, which are all valid dependencies.
A low number of invalid dependencies were found. All of which are likely caused by multiple levels of abstraction and transitive dependencies. Examples include StatisticsReportTestBase.java's dependency with Timestampdata.java. No direct function calls were found; we suspect that there may be a transitive usage through a utility class, DateTimeUtils which is a static helper class, but there is no direct dependency between the two classes.
Validity rating: 95.8%

Dependencies only extracted by Import:
Total size: 2663, Sample size: 93.
Number of invalid dependencies found: 38
Explanation: Within the sample that were examined, we found that due to how Apache Flink utilizes Maven to manage some of the exterior dependencies, the method included a portion of these due to its parsing method of looking for org.apache.flink.* packages imports, causing org.apache.flink.shaded.* dependencies to also be accounted for. All the invalid dependencies were caused by this error. An example is WritingMetadataSpec.java which has 4 dependencies mapped to external Json libraries which are not under the scope of this study's concern.
Validity rating: 59.1%

Dependencies only extracted by srcML:
Total size: 10629, Sample size: 95.
Number of invalid dependencies found: 49
Explanation: Within the sample that were examined, we found that due to the method's parsing for function calls and class-level dependency extraction, the method included a portion of dependencies made to other apache libraries, including calcite, used on sql operations. For example, 2 dependencies were mapped to calcite libraries in SqlReset.Java. These are dependencies that are excluded from this study's scope.
In addition, because the method extracts dependencies from class level function calls. There are enums and inner classes which are being extracted as dependencies. For example,

MemorySize.java includes the enum MemoryUnit, and dependencies were extracted based on the enum, effectively causing an invalid dependency of a class file upon itself.
Validity rating: 48.4%

Dependencies only extracted by Understand and srcML:
Total size: 4974, Sample size: 94.
Number of invalid dependencies found: 0
Explanation: Within the sample that were examined, all dependencies that belong to this group are valid dependencies similar to the sample group identified by all three methods; there are strong connections between the dependent and the depended. In this scenario, all the dependencies were not discovered by the Import method, because Import statements were not needed in these cases as the two class files are located within the same package. Interestingly, we did not find any sample where the package path is explicitly declared within the source code causing Import parsing to miss the dependency.
Validity rating: 100%

Dependencies only extracted by Understand and Import:
Total size: 38115, Sample size: 96.
Number of invalid dependencies found: 11
Explanation: Within the sample that were examined, we found that the majority of the dependencies that were found by Understand and Import are ones where a high level of abstractions is involved, either through multiple levels of interface implementation or inheritance. There are no direct function calls and or implementations, hence these were not extracted by srcML. However, these dependencies appear all to be valid (high number of unique implementations of classes such as tuple.java, types.java).
A small number of dependencies were based on annotation classes that do not serve any concrete dependency purposes and were deemed invalid. An example is dependencies to Public.java, which is an annotation class defined by Apache Flink developers to simply mark classes as public, stable interfaces.
Validity rating: 88.5%

Dependencies only extracted by Import and srcML:
Total size: 3031, Sample size: 93.
Number of invalid dependencies found: 55
Explanation: Within the sample that were examined, we found that the majority of dependencies not extracted by Understand are ones which are not file-level dependencies. For example, the dependencies to Keymode and Valuemode are extracted by both srcML and Import. These are two enum classes declared under TestData.java which is one that is detected by Understand. Hence, these duplicate dependencies, despite being existing dependencies, are deemed invalid in our case for consistency.
There are also cases of shaded external packages dependencies which are also treated as invalid.
Validity rating: 40.9%

Dependencies retrieved only by:	Count	Validity in %
Understand	26951	95.8
Import Parsing	2663	59.1
srcML Parsing	10629	48.4
All 3 Methods	48753	100
Understand and Import	38115	88.5
Understand and srcML	4974	100
srcML and Import	3031	40.9

6.3 Precision & Recall

Based on the results of section 6.2, we can calculate the precision and recall rate of the three different methods (https://en.wikipedia.org/wiki/Precision_and_recall).
All relevant instances = 26951 * 95.8% + 2663 * 59.1% + 10629 * 48.4% + 48753 + 38115 * 88.5% + 4974 + 3031 * 40.9 % = 121235

Method 1: Understand:
Relevant retrieved Instances = 26951 * 95.8% + 48753 + 38115 * 88.5% + 4974 = 113277.
All retrieved Instances = 118793
Precision: 0.9536 / 95.36%
Recall: 0.9344 / 93.44%

Method 2: Import Parsing:
Relevant retrieved Instances = 2663 * 59.1% + 48753 + 38115 * 88.5% + 3031*40.9% = 85298.
All retrieved Instances = 92562
Precision: 0.9215 / 92.15%
Recall: 0.7036 / 70.36%

Method 3: srcML Parsing:
Relevant retrieved Instances = 10629*48.4% + 48753 + 4974 + 3031*40.9% = 60111.
All retrieved Instances = 67387
Precision: 0.8920 / 89.20%
Recall: 0.4958 / 49.58%

Based on the calculated precision/recall, we can observe that the Understand method performed the best in extracting the valid dependencies in both quantity and quality. Import parsing was also able to obtain a decent precision (implying that the Apache Flink developers follow a consistent practice of using Import statements, and package organization), but failed to extract ~30% of valid dependencies (ones that exist within the same package). srcML performed the worst in terms of both precision and recall.

7. Limitations

The current approaches involve human judgement and manual processing when it comes to sample analysis and inference. As a result, judgments were made based on previous knowledge of Apache Flink architecture.

The inference confidence interval and level are also not set to a very high number due to restraints on human resources and time. A higher confidence interval (10 was used) would certainly yield more accurate findings.

As for srcML, it is not actively maintained, the last commit was in 2021 and this is a large barrier for this software. Its downloads are all for old versions of operating systems. Additionally, the output by itself isn't very helpful for analysis.
In addition, our implementation of srcML parsing is a class-level dependency extraction as it is our intent to utilize a different approach for method 3 and to focus on method calls and constructor usages. This led to noticeable differences as outlined in quantitative and qualitative analysis as the other two methods relied on file-level dependencies.
It may be worth the time and effort to utilize Understand's class-level dependency analysis to observe if there are increased commonalities.

All the custom scripts and programs implemented to conduct this analysis lack thorough testing. Limited number of test cases were used and all were manually verified. If given sufficient time, the code should be thoroughly tested against all edge cases to ensure there are no mistakes in parsing and comparison logic.

8. Technique Summary

For the first technique: Understand, we found that it is a very comprehensive dependency extraction method as it provides a high number of dependencies per entity and is detailed. Ultimately, its few inaccuracies result from how the tool extracts dependencies that are highly transitive and may not be representative of actual dependencies.

For the second technique Import Parsing: It is a comparatively Simplistic and Barebone technique and it provides a decent representation of the dependencies. However, the technique is ultimately reliant solely on the usage of Import statements, which is not a reliable source of dependency extraction. As seen in our analysis, there are a number of valid dependencies that are not established using import as it is unnecessary to explicitly declare the package path. In Apache Flink's case, the developers consistently use Import statements, but in some other projects, it is not for certain that Import statement will be properly utilized (no redundant usage, and no explicit package path declaration in the code without import), which will lead to significant negative impact on the recall rate of this extraction technique.

srcML: is a class-level extraction technique and it identifies a high number of entities. However, when analyzed against file-level extraction techniques, many duplicate dependencies (enums and inner classes) are extracted. In addition, this technique is heavily influenced by parsing script logic. In this case, we parsed for method calls, method parameters and constructor usage,

but as shown in the analysis section, the dependencies extracted will miss a large number of valid dependencies that may be transitive or abstracted, yet which are important.

9. Lessons Learned

Lesson 1: Each technique has pros and cons that make one choice better than another.

Lesson 2: Tools may be outdated and not maintained, in particular Jarviz is one example of this. Alternatives had to be used as a result. srcML also has concerns with up-to-dateness.

Lesson 3: Tools may have bugs as seen in jdeps that lead to highly inaccurate extractions.

Lesson 4: Comparisons between class-level dependencies and file-level dependencies is fundamentally not viable and undermines the assessment of different techniques. srcML could be much more accurate than it is represented in our study while Understand file-level extraction could have much worse precision/recall if compared with an alternative effective file-level extraction method.

Lesson 5: Many of our results are in line with the shortcomings or cons we identified with each technique. For example, Import parsing's reliance on accurate Import statements. Therefore, having a good understanding of the available methods and the goal of the extraction can help determine which method to use in the very beginning without having to test all the possible methods.

Conclusion

This report provides an examination of various dependency extraction methods applied to the Apache Flink software, using Understand, jdeps, and srcML tools. Our findings reveal that each tool possesses unique strengths and weaknesses, which impacts the nature and quality of the dependency extraction. Through extraction processes and quantitative analysis involving precision and recall metrics, we gained insights into the capabilities and limitations of each method. This report demonstrates the importance of selecting the appropriate tool based on specific project requirements and constraints. The limitations of each strategy, as discussed, influence the outcomes of this analysis and should be taken into consideration when analyzing software architecture.

References

[1]https://github.com/ExpediaGroup/jarviz
[2]https://support.scitools.com/support/solutions/articles/70000583144-how-dependencies-in-understand-are-determined
[3]https://www.srcml.org/#download
[4]https://docs.python.org/3/library/xml.etree.elementtree.html

https://en.wikipedia.org/wiki/Precision_and_recall
https://www.surveysystem.com/sscalc.htm

Appendix

1: C#: Get Import Dependencies:
https://raw.githubusercontent.com/dzinkiv/eecs4314/main/Assignment-4/C%23/GetImportDependencies.cs
2: Quantitative/Qualitative(sampling) analysis program:
https://raw.githubusercontent.com/dzinkiv/eecs4314/main/Assignment-4/C%23/Program.cs
3: Analysis Process Diagram:
https://github.com/dzinkiv/eecs4314/blob/main/Assignment-4/Analysis%20Process.png