Detecting Cross-language Dependencies Generically

Theodoros Polychniatis
Jurriaan Hage
and Slinger Jansen
Utrecht University
The Netherlands

Eric Bouwers[*]
and Joost Visser[‡]
Software Improvement Group
[*]*Delft University of Technology*
[‡]*Radboud University Nijmegen*
The Netherlands

Abstract

In order to evaluate large, heterogeneous information systems (i.e., comprising modules developed in diverse programming languages) a method to detect dependencies among these modules is needed. Although there is a variety of methods that can detect dependencies within a single programming language, the available cross-language detection methods use extensive language specific information to parse and analyse modules written in different languages.

In this paper, a new method for detecting cross-language dependencies is proposed. This method is generic, yet accurate and can support new languages with minimal effort. To evaluate the method, a tool was created and a series of experiments was conducted on a small case study for which dependencies had been extracted manually. The evaluation shows that the method is effective, extensible and easily explainable.

1. Introduction

The modification of heterogeneous software systems is a difficult and complex task. Before any modification to a software module is made, an impact analysis is carried out to evaluate the quality of the system and identify possible risks in the module's dependencies. It is hard to identify cross-language dependencies without manual inspection because the dependencies among heterogeneous modules are difficult to trace. Such dependencies can take many different syntactical forms, can be dynamic rather than just static, and may be indirect via frameworks, databases, or scripts. The detection of dependencies becomes even more challenging due to the large number of different languages and their possible combinations.

To have a better understanding of the requirements that a cross-language dependency detection method should have, we conducted six half hour, semi-structured interviews with experienced software evaluators. All evaluators work as consultants at the Software Improvement Group (SIG), a company that is routinely involved in quality evaluations, for at least two years. According to the consultants, the top three requirements for the method are:

1) *Accuracy* - the ability to extract the correct dependencies.
2) *Explainability* - the ability to be explained to non-technical personnel.
3) *Extensibility* - the ability to be extended to include an unsupported language (combination).

Although there is a lot of literature in evolutionary or dynamic dependencies, we focus our research on static analysis of single snapshots of systems because it is not always possible to execute the code or have older versions of it. The majority of the state of the art methods in cross-language dependency detection use meta-modeling techniques. These techniques involve fully or partially parsing the source code modules using the grammar and syntax of a language, and then creating models out of them. These models adhere to a common, language independent meta-model that enables querying of the models' elements in a uniform way. Queries are then used to extract the dependencies between modules. Moose [1], Eclipse MoDisco[1] and GenDeMoG [2] are representative projects that take advantage of these meta-modeling techniques.

On the other hand, DSketch [3] (also known as GrammarSketch) does not use meta-models to detect dependencies. It provides an easy interface language for writing patterns for matching modules instead of writing regular expressions directly. However, the method is language specific and an API for supporting new languages is not provided.

1. Eclipse-MDT MoDisco: http://eclipse.org/MoDisco

Because the reviewed methods focus mainly on the accuracy of their results, they use an inflexible, language specific approach which impacts their extensibility. To the best of our knowledge, there is no scalable, robust, general approach for cross-language dependency detection. Our goal is to create a method for detecting dependencies among software modules, focusing on three requirements: accuracy, explainability and extensibility.

2. Terminology

In the information retrieval literature, commonly accepted metrics for measuring *accuracy* are [4]:

- **Recall**: the fraction of relevant documents that are retrieved.
- **Precision**: is the fraction of retrieved documents that are relevant.

Wilde gives the following definition for dependency: "*From the point of view of the maintainer, there is a dependency between two components if a change to one may have an impact that will require changes to the other*" [5]. We extend Wilde's definition by classifying dependencies according to the type of error that is possibly caused by a change in one of the dependent modules:

- *Direct* - Strong dependencies where a possible change can cause compilation errors.
- *Framework* - Weaker type of dependencies which would probably cause runtime errors.
- *Indirect* - Even weaker dependencies, which may cause errors in functionality.

The classification was introduced because it defines dependencies in more detail, and this makes the validation process, which involves manual detection of dependencies, more accurate.

3. Solution

We propose a generic approach that trades off accuracy in favor of extensibility and explainability.

The interviewees from SIG stated that, although they tolerate relatively low numbers of retrieved, actual dependencies (i.e. True Positives - TP), they are more interested in having a minimal number of falsely detected dependencies (i.e. False Positives - FP). This is because the retrieved dependencies have to be trustworthy. However the TP should not too small.

As for the *extensibility*, they mentioned that the method should be able to support a new language in less than two days. For the *explainability*, they said that the algorithm used in the method should be explainable to a non-technical audience. The interviewees also expressed their interest in detecting cross-language dependencies that involve database access, web services and configuration artifacts used to bind different technologies within a system.

```
create a graph of the system's source modules
for each module in the graph do
    replace comments and special characters in the module's content with spaces
    tokenize the text by splitting it on white space
    extract all the tokens (words)
    for each token do
        check the graph for a node with the name of the token
        if token-node does not exist then
            create a node with the same name as the token
        end if
        create an edge from the module to the token-node
    end for
end for
for each token-node do
    if node is connected with fewer than two modules from different languages then
        remove node
    end if
end for
```

Figure 1: Core algorithm of the token based approach

3.1. Core algorithm

The core algorithm of the proposed method (see Fig. 1) is a generic, naive approach based on matching common tokens between two possibly dependent source modules. The results of this algorithm are improved by applying a set of statistical filters.

The following details of the algorithm were omitted because we want to provide just the basic idea while keeping it readable in this document:

- A special character is any character except letter, number and underscore.
- Strings that consist only of numbers are omitted in tokenizing.
- Tokens with only one character are omitted.

Note that the algorithm does not classify dependencies (see Sec. 2). This is not seen as problematic since all types of dependencies can lead to errors in functionally.

3.2. Statistical filtering

As expected, the above algorithm creates a large number of FP. The next paragraphs describe the steps performed to improve the accuracy of our algorithm.

Filtering out frequent tokens. An observation that was made from looking at the source modules is that reserved words (e.g., *false, true, int*, etc.) appear with high frequency as compared to non-reserved words. This led to the creation of the first statistical filter which calculates all the frequencies of unique tokens per language within the system. This is done by creating a map of all the unique tokens that are extracted from all modules per programming language, and by counting the number of times each token is found. The input to the filter is a frequency threshold, which means that tokens that appear with higher frequency than the threshold value, are removed, and are not involved in the detected dependencies.

Filtering out frequent tokens per module, language and language category. Occasionally the source modules contain a large number of statements, which do not necessarily include language specific tokens. An example that reveals this problem is a system that has many large Java files. The words *class, interface, extends, implements* will then not have a high frequency, hence they will not be removed.

To overcome this problem, the *Tokens per modules filter* is introduced. This filter calculates the frequency of the tokens which appear in different modules of the same language. It collects all the unique tokens for the specific language and then for each token it counts the modules containing it. The filter calculates for each language, and for each token the fraction:

$$\frac{\#modules\ in\ a\ language\ that\ contain\ the\ token}{Total\ \#modules\ in\ this\ language}$$

The tokens that appear with a frequency higher than the threshold, are filtered out.

Some languages have derivatives or multiple usage. E.g., the C-type languages (C, C++, C#) or the numerous XML-type languages. Although it is useful to treat these cases separately, as different languages, they share common characteristics. Therefore, the concept of language category was introduced to cover languages with similar characteristics (notation, grammar and syntax). As a consequence, a derivative of this filter was created which calculates the same fraction as above but per language category instead of per language.

Weight filtering. Another observation that was made is that some FP were caused by dependencies where only a small number of tokens were matched from the two dependent files. Examples of such tokens are *in, of, id, encoding, type*, etc. To remove these FP, the concept of *"weight"* as a property of a dependency was introduced. The weight equals the number of tokens which are matched in the two modules that the dependency refers to. Subsequently, a filter was created that filters out dependencies with low weight. It gets as input a threshold, i.e., a number, below which every dependency is removed.

Attributes	**Value**
Direct dependencies	6
Framework dependencies	53
Indirect dependencies	35
Total relevant dependencies	94
Non-dependent file combinations	306
Collection (*All possible file combinations*)	400

Table 1: Ground truth data

4. Validation

To evaluate the validity of our approach, the proposed method is executed and evaluated against the three most important criteria as discussed in section 1. The method is created through an iterative process of brainstorming for algorithms, implementation, testing, evaluating and refinement. Within each iteration we performed the following steps:

1) Apply the method on the case study project.
2) Retrieve the results.
3) Compare the results with the ones retrieved with a ground truth method (GTM).
4) Calculate the accuracy metrics (see sec. 2).
5) Examine the FP and the non-retrieved dependencies to improve the algorithm.
6) Judge the algorithm in terms of explainability and extensibility.

A sample application from the Spring webflow framework or SWF[2] was selected as the case study because it is multi-lingual, open-source, and small. Therefore the dependencies could be extracted manually in a reasonable time-frame. The project contains a diverse set of languages such as Java, Java Server Faces, SQL and Spring specific XML languages (for DB access, web-flow and dependency injection).

Manual checking was selected as a GTM as opposed to using one of the existing methods. This is because none of the reviewed methods had documented measurements of both metrics of accuracy. The GTM was applied on the case study taking into account that:

- if two modules are dependent in both directions, this dependency is counted once;
- dependencies among modules of the same language are filtered out.

Table 1 displays the information retrieved with a manual check for dependencies. The terms in the table are explained in sec. 2.

To evaluate more precisely the extensibility of the algorithm a second experiment was performed where the method was extended to cover an unsupported language. The high explainability of the the method was confirmed after presenting it to 30 SIG employees.

2. http://www.springsource.org/spring-web-flow

Recall	Precision	Parameters
100%	11.6%	Core algorithm, no filters applied
94.7%	37.9%	Freq.$_{token/modules/language}$:60%, Freq.$_{token/modules/category}$:55%, Min. weight:0
85.1%	48.2%	Freq.$_{token/modules/language}$:60%, Freq.$_{token/modules/category}$:25%, Min. weight:0
39.4%	80.4%	Freq.$_{token/modules/language}$:70%, Freq.$_{token/modules/category}$:30%, Min. weight:4
16%	93.8%	Freq.$_{token/modules/language}$:35%, Freq.$_{token/modules/category}$:25%, Min. weight:4

Table 2: Best results of the algorithm

5. Results

Multiple experiments were conducted with the possible combinations of thresholds for the above filters. In Table 2 we display the combinations that based on the two accuracy metrics are most interesting.

The frequent tokens filter is missing from Table 2 because the best results in accuracy were achieved without applying it. From the table we observe that it is difficult to achieve a parameterization which results in very good values for both recall and precision. However, depending on the desired priority of the accuracy aspect, a high value in the corresponding metric can be achieved, with a trade-off in the other. In our case, 94.7% of the dependencies can be captured, with a precision of 37.9%. On the other hand, if precision is the top priority, an 80.4% precision can be achieved while capturing 39.4% of the dependencies.

With this research it can be shown that a very simple, generic and explainable algorithm can be used to detect dependencies with a minimal input of language specific information.

To generalize our findings, the method was executed on a governmental, large scale system (1MLOC), using the parameterization that resulted in 80.4% precision. The main languages of this service based system were Java, XSD and WSDL. The method had to be extended to support WSDL. The time for implementing the extension was less than three hours. The recall could not be calculated because the dependencies could not be manually extracted in a reasonable time-frame, but the precision was found to be similar (76.7%). Nevertheless, more systems are needed to be able to safely generalize the results.

6. Conclusion

In this paper, we introduced a simple, generic method for detecting cross-language dependencies, steering away from the traditional, inflexible code-parsing methods. Although there is room for improvement the results are promising. Specifically the following contributions are made:

- Identified the need for a generic cross-language dependency detection method that has high precision and is easily explainable.
- Developed a solution for the core algorithm and the initial filtering approaches.
- Evaluated the solution with a predefined evaluation protocol.

Based on these results we define three areas of future work. First, we envision the creation of a benchmark to extract pre-defined parameterizations. For such a benchmark, a classification of systems according to their language set and size is needed. The proposed method can then be applied and validated, and if the parameters are similar the best parameterization for a given category of systems can be extracted.

Second, the behavior of the method can be investigated at the component level. In order to do that, the module level dependencies need to be aggregated to this level. It may be possible to increase the accuracy of the method by applying other type of filters on the components or their dependencies.

It is also possible to improve the accuracy with the application of more statistical filters, e.g., a filter that removes tokens with fewer than a specified number of characters. Research can be conducted in utilizing physical language dictionaries, similarity algorithms and other text mining techniques to achieve better results.

References

[1] S. Ducasse, T. Girba, A. Kuhn, and L. Renggli, "Meta-environment and executable meta-language using smalltalk: an experience report," *Software and Systems Modeling*, vol. 8, no. 1, pp. 5–19, 2009.

[2] R.-H. Pfeiffer and A. Wasowski, "Taming the confusion of languages," in *Proceedings of the 7th European conference on Modelling foundations and applications*, ser. ECMFA'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 312–328.

[3] B. Cossette and R. J. Walker, "Dsketch : Lightweight , adaptable dependency analysis," *Change*, pp. 297–306, 2010.

[4] C. Manning, P. Raghavan, and H. Schuetze, *Evaluation in information retrieval*, online edi ed. Cambridge UP, 2009, no. c.

[5] N. Wilde, "Understanding program dependencies," *Program*, no. SEI-CM-26, 1990.