Utrecht University

Master Thesis
ICA-0508861

Finding Similar Software with Ontology Matching Techniques

Tim Breedveld

supervised by
Dr. Wishnu Prasetya
Dr. Jurriaan Hage

July 10, 2018

Abstract

With the current availability of open source projects, the reuse of existing test cases might be a cost-effective way to reduce time spent on creating new test cases. In order to reuse existing test cases, suitable candidates first need to be located. The approach in this study proposes to first find code components that have the same functionality and then locate test cases covering these code components.

In order to find semantically similar code components *ontology matching techniques* are used. An ontology can be defined as a vocabulary which is used to describe a specific domain and relationships that exist among the concept of that domain. It tries to focus on concepts and semantics rather than the syntax of a message or a system. To capture the semantic information of a code component in an ontology, a method is presented to extract ontologies from a source and target software project. These ontologies are used as input for the ontology matching framework *AgreementMakerLight* (AML) which produces a set of mappings between the classes of the source and target ontology. These mappings can be processed to determine which components units are similar to each other.

Two case studies were conducted to demonstrate the effectiveness of this technique. It can be concluded that ontology matching techniques can effectively be used to find semantically similar software at a class level.

Contents

1 Introduction

The motivation for this study is to advance the field of software testing. Software testing is widely recognized as a critical process to determine the quality of a software item [1–3].

An important part of software testing is writing and maintaining unit tests for the smallest testable pieces of software. These unit tests can either be created manually by a software developer or automatically by a tool. Writing unit tests manually is a very time-consuming process and is therefore often neglected or skipped entirely. Even when unit tests are written, maintaining the unit tests in a meaningful way can be a challenging task. A survey conducted under software developers indicates that most developers are convinced that testing improves software quality but that they do not enjoy writing unit tests and rather spend time coding new features [4]. Therefore one of the ultimate goals in software testing is the 100% automatic generation of unit tests [2].

A major challenge for automatic test generation is selecting the best inputs from an almost infinite domain of inputs (where the definition for 'best' can vary depending on the goal of the test) while only limited computational power is available [5]. Techniques like random testing [6, 7], model-based testing [8, 9] and search based testing [10] are promising, but fully automatic test generation is not yet possible with the current level of research and computational power available.

What if we could reuse existing test cases instead of generating new test cases? With the popularity of open source software and the numerous public repositories accessible via the internet it might be possible to reuse existing test cases written for similar pieces of software in other projects. The concept of reusing software has been studied at length and has been applied by the industry with great success. It reduces development time and improves overall quality of the software. However the practice of reusing test cases, unit tests or other artifacts from the testing process is not widespread. This leads to the main research question:

RQ 1: *Is it feasible to automatically reuse test cases?*

In order to automatically reuse test cases, first test cases that are suitable for reuse need to be identified. When an existing test case is found, it is very likely that it will require some changes before it can be used. Depending on the method used to find the test cases, they might need to be translated into a different programming language or transformed in some way to make

them work.

RQ 1.1: *How can we automatically find existing test cases that can be reused?*

RQ 1.2: *How can we automatically transform existing test cases so they can be reused?*

An approach to finding these existing test cases it to search for units of code in software that are 'similar' to each other. Similar in the way that both units of code that have the same functionality and thus are semantically similar. For example reading and writing of a file or the validation of a password. It can be assumed that there is a high probability that test cases for semantically similar units of code also share a high degree of similarity.

RQ 1.1.1: *How do we automatically identify semantically similar units of code?*

RQ 1.1.2: *How do we find tests that cover these units of code and select test cases that can be reused?*

The research questions mentioned above are too much of a challenge for a single master thesis. This thesis will therefore focus on answering RQ 1.1.1.

Finding similar software has been investigated by many researchers for different practical purposes like duplicate code detection [11–15], plagiarism detection [16], virus detection [17] and image recognition [18] However, almost all approaches focus on finding code with a similar *syntax* instead of a similar semantic meaning. And most approaches that do focus on semantic meaning required a high degree of human interaction [19–21].

The focus of this research is to see if it is feasible to automatically find semantically similar software. To do this, the semantic meaning of a software component will be captured in an *ontology* and compared with other ontologies using ontology matching techniques.

This thesis is organized as follows. First, in section 2, ontologies and ontology matching are explained in greater detail. In section 3, related literature is discussed and in section 4 the research method is stated. Section 5 will go into detail about the ontology matching process and section 6 and 7 contain the results of two case studies. Finally, section 8 will list the conclusion and future work.

2 Ontology matching

An ontology can be defined as a vocabulary which is used to describe a specific domain and relationships that exist among the concepts of that domain. It tries to focus on concepts and semantics rather than the syntax of a message or a system. It is a technique that is one of the cornerstones of the semantic web, as it allows the semantics of the current world wide web to be expressed in a format that can easily be processed automatically [22].

2.1 The Web Ontology Language

The *Web Ontology Language* (OWL) is the international standard recommended by the World Wide Web Consortium (W3C) for communicating ontologies. OWL extends the older Resource Description Framework (RDF) and RDF schema and is expressed in the extensible markup language (XML).

In OWL the most basic concept is that of an *individual*. An individual is a member of one or multiple *classes*. Classes can be used to group individuals with similar characteristics. It is possible to create taxonomies using *subClassOf* to relate a specific class to a more general class. All individuals in OWL are a member of class *Thing*.

The example below defines the class `Person` as a subclass of `Thing` and specifies the individual `John` to be a member of class `Person`. The `rdf:about` attribute contains the unique identifier for the class. The example also shows the use of `rdfs:label` *annotation*. In this example a label is added to provide a human-readable descriptive name for each class.

```
<owl:Class rdf:about="http://class.example.org#Person">
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
    <rdfs:label xml:lang="EN">Person</rdfs:label>
</owl:Class>

<owl:NamedIndividual rdf:about="http://class.example.org#John">
    <rdf:type rdf:resource="http://class.example.org#Person"/>
    <rdfs:label xml:lang="EN">John</rdfs:label>
</owl:NamedIndividual>
```

Properties can be used to assert facts about the members of classes and about individuals. OWL has two main property types: properties that specify a relation between two individuals are called *object properties* and properties that specify a relation between an individual and a value are called *datatype properties*. Properties can also be expressed in taxonomies.

The example below defines the object property `isFatherOf` and the datatype property `hasBirthDate`. It also specifies that the range of values for the `hasBirthDate` property must be of type `dateTime`.

```
<owl:ObjectProperty rdf:about="http://class.example.org#isFatherOf">
    <rdfs:subPropertyOf rdf:resource="http://www.w3.org/2002/07/owl#topObjectProperty"/>
</owl:ObjectProperty>

<owl:DatatypeProperty rdf:about="http://class.example.org#hasBirthDate">
    <rdfs:subPropertyOf rdf:resource="http://www.w3.org/2002/07/owl#topDataProperty"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#dateTime"/>
</owl:DatatypeProperty>
```

The following example declares that the individual `John` has a relation of type `isFatherOf` with individual `Sarah` and that `John` has a birthDate of type dateTime with the value of 30 May 1950.

```
<owl:NamedIndividual rdf:about="http://class.example.org#John">
    <rdf:type rdf:resource="http://class.example.org#Person"/>
    <isFatherOf rdf:resource="http://class.example.org#Sarah"/>
    <hasBirthDate rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
        1950-05-30T09:00:00
    </hasBirthDate>
</owl:NamedIndividual>
```

For a complete introduction into OWL see the paper from Jeff Heflin [23] and the OWL W3C reference[1].

2.2 Ontology matching

Because of the distributed nature of the world wide web more and more ontologies are being developed and many of them describe similar domains. To allow querying or communication over multiple ontologies it is required to create a mapping between these ontologies. Such a mapping is called an *alignment* and specifies how to map a concept in the source domain to one or multiple concepts in the target domain. To create an alignment, *ontology matching* is performed. Ontology matching is the process of finding all matching concepts in two ontologies and selecting the best candidate to use. The figure below is a simple example of a match between two ontologies. Nodes with the same colors are matching nodes.

This simple match already contains some interesting features. The match between the SSN nodes looks trivial but to establish this match a matcher needs to identify that the relation between person and SSN is similar to that of Human to SSN, to validate that the SSN nodes do mean the same

[1] https://www.w3.org/TR/owl-ref/

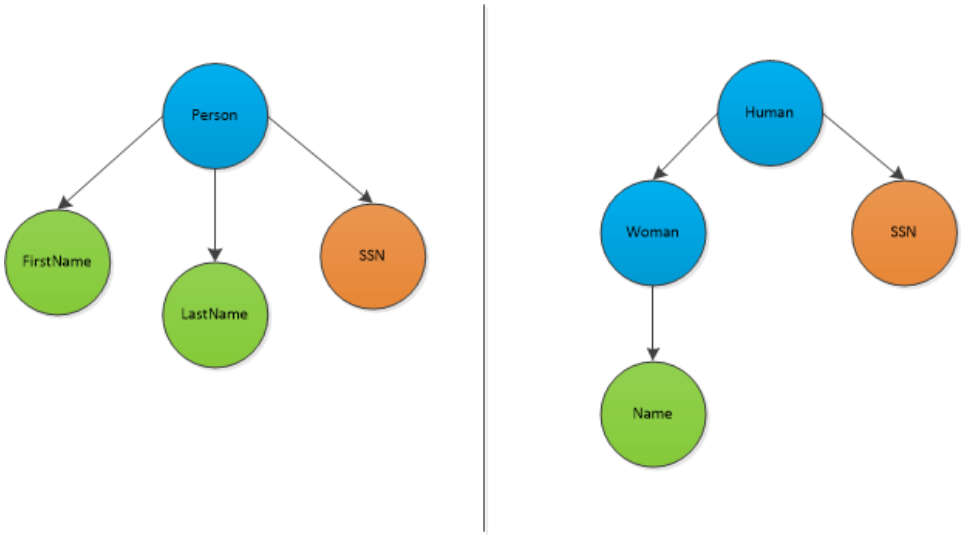

Figure 1: Example of an ontology match. Nodes with the same colors are matching nodes.

thing. The match between FirstName + LastName and Name requires not only investigating the nodes themselves but also investigating sibling nodes. Finally, the match between Person and Human + Woman requires studying related nodes and an interpretation of the labels of the nodes. The fact that a Person is similar to a Human + Woman is not a trivial match to make. This requires the matcher to know that these entities are semantically similar although they are structurally different.

Ontology matching is a form of schema matching and much research has been devoted to these areas [24–27]. As a result, many matching techniques have been researched and tested. Below is a classification of the different matching techniques in use today. Otero-Cerdeira et al. provide a very extensive overview based on the work of Euzenat et al. [28] and this overview is largely based on their work [29].

First of all, ontology matching techniques can be classified by the interpretation of the input information.

- *Element level matchers* match elements in an ontology in isolation
- *Structure level matchers* match elements based on their positions and relations to other nodes in an ontology

The techniques can further be classified in:

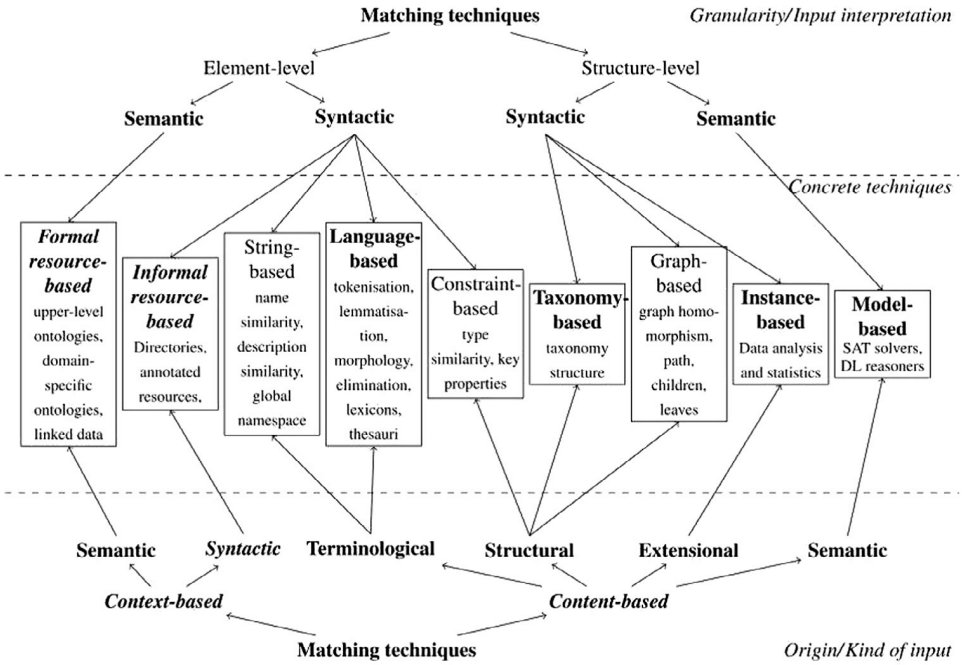

Figure 2: Ontology matching techniques classification. Extracted from [28]

- *Syntactic matchers* limit the input interpretation to the instructions in their corresponding algorithms
- *Semantic matchers* use formal semantics to interpret their input and justify their results

Matching techniques can also be classified by the type of input that they use:

- *Content based matchers* focus on the information provided by the ontologies being matched
- *Context based matchers* use not only the ontology but also external resources

These matching techniques can be subdivided in:

- *Terminological matchers* consider their input as strings
- *Structural matchers* use the structure of the ontologies

- *Extensional matchers* consider the instances of the classes
- *Semantic matchers* use semantic interpretation of the input

Many matchers use hybrid or combined approaches or use multiple pass strategies to improve their results based on what was learned from the previous pass. These multiple pass approaches often use reasoning based techniques to infer extra information that can be used in the next pass.

2.3 Alignment Quality

To evaluate the quality of an alignment it is required to compare it with a *golden standard*. This golden standard is the perfect alignment and is often created manually. Alignment are compared on *precision* and their *recall*. The precision or *positive predication value* is the fraction of the relevant matches among the retrieved matches. Precision can be expressed by the formula:

$$precision = \frac{\text{true positive}}{\text{true positive} + \text{false positive}}$$

A true positive is a match found by the matching process that is actually a correct match. A false positive is a match found by the matching process that is not a correct match. The result of the formulas is a number between 0 and 1, where 1 means that all results found are correct matches and 0 means that all results found are not correct.

Recall or *sensitivity* is the fraction of relevant matches found over the total number of relevant matches available.

$$recall = \frac{\text{true positive}}{\text{true positive} + \text{false negative}}$$

A false negative is a match that was available but was not found by the matching process. The result of this formula is a number between 0 and 1 where 1 means that all available matches have been found by the matches and 0 means that the matches failed to find any match.

Both precision and recall can be combined in the F-measure. This is the harmonic mean of both values and can be expressed by:

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

For example: there are two ontologies for which ten matches can be made between elements in those ontologies. A matching process is able to find 6 matches: 4 are actually good matches and 2 matches are incorrect matches. The precision for this matcher is: 0.67, the recall for this matcher is: 0.4 and the F_1 score is: 0.5.

3 Related work

3.1 Reusing test cases

The concept of reusing software has been studied at length and has been applied in the industry with great success. It reduces development time and improves the overall quality of the software. However, the practice of reusing test cases, unit tests or other artifacts from the testing process is not widespread. This is despite the fact that the oldest paper on this subject already dates from 1994. Von Mayrhauser et al. uses a model-based testing (or domain based testing) approach that allows testers to create parameterized test templates and generate test instances from these templates. The templates could be reused to create different tests throughout the lifespan of a projects. They created the tool "sleuth" that was successfully used on an industry project. Besides increased test capabilities they reported an increased productivity for testers [30]. However, the creation of generic test templates requires more work and attention than the creation of regular test cases. A study that examined the effectiveness of a repository with generic text-based test cases shared by human testers, determined that around 26% of test cases could be reused although small changes were required to reuse a test case in a new project. They concluded however that the benefit of reusing test cases did not outweigh the extra amount of work required for writing more generic tests cases and using the repository [31].

The concept of using ontologies to reuse test components has also been explored in literature. Most techniques express test artifacts in an ontology and use ontology matching techniques to find similar artifacts [19–21]. Li and Ma wrote a paper that provides a formulation of test cases in an ontology and a semantic similarity method based on wordnet to retrieve such test cases from a repository. By incorporating wordnet into their method they were able to find concepts that were semantically similar instead of finding concepts that were only structurally similar. It also includes a method to transform test cases so can they can be run in different environments. A small-scale experiment showed the feasibility of the proposed method and yielded promising result [32]. Unfortunately, all approaches mentioned above depend heavily on human interaction in either creating the ontologies, or searching the repositories for ontologies. As such these approaches can be considered impractical and too slow for day to day use by software developers.

3.2 Finding semantically similar software

The original goal of this research is to automate the reuse of existing test cases. However, as we can only do so much in a single thesis, the main focus of this study is finding semantically similar software.

Finding similar software has been investigated by many researchers for different practical purposes like duplicate code detection [11, 13], plagiarism detection [16], virus detection [17] and image recognition [18]. However, most of the techniques used in this area focus purely on syntax. Smeureanu and Lancu propose a method focused on finding plagiarism in source code using ontologies. It extracts ontologies from the source code structure and uses SPARQL queries to generate a set of metrics over the ontologies. These metrics can be compared to detect similarity [33]. However, the ontologies used by Smeureanu and Lancu only contained information about the program structure (e.g. the amount of while loops and if statements) and did not store any semantic information.

The concept of using semantic information was initially mainly used for document comparison. Si et al. present a method for detecting semantic similarities in Latex documents [34]. First, they capture the semantics of a document by extracting keywords. After this, they create a tree structure of the document using the sections and subsection and assign keywords that correspond to each section node in the tree. This data is stored in a repository and used to match new Latex documents against existing documents. As their goal is to uncover plagiarism, their method of comparison is rather strict and does not leave room for subtle changes. Shenoy et al. proposed to extract ontologies from documents using an ontology extractor and compare these ontologies using an ontology mapper that focusses on semantics rather than structure. They were able to detect texts that have a different structure, but the same semantic meaning [35]. Unfortunately, the paper did not include any information on the mapper implementation or the accuracy of the proposed method. A recent study conducted by Aydemir et al. uses natural language processing techniques to extract semantic information from models representing different views on a single domain. They created a framework that was capable of identifying similar concepts among these models and also identify missing concepts in a model [36]. Although this research did not explicitly use ontologies, it shows a clear benefit of using semantics over syntax. Because they focused on semantics rather than syntax, they were able to ignore the different modeling languages used to create the models.

Currently, the value of semantic information in source code is becoming

more evident. Multiple studies have used this semantic information to help human developers understand large software projects. Kuhn et al created a technique that uses semantic information to characterize classes and projects [37] and Ieva et al. uses both semantic and structural information to identify the most important components in a software project [38].

4 Method

To answer the question whether it is feasible to automatically find semantically similar software we will first need to extract ontologies containing semantic information from software projects. When this is done, existing ontology matching techniques can be used to create an alignment between these two ontologies. If such an alignment can be established we can conclude that these ontologies are similar to some extent and thus that the software projects used to extract the ontologies are also similar to some extent. This approach will be tested with two case studies.

The first case study will try to match classes in consecutive versions of a project. The Apache Commons Collections [2] source code will be used for this experiment. This open source software initiative has a large codebase, 16 releases and 5 major version updates. It also has an excellent issue tracking system that can provide background information on the changes in different versions. By using the matcher on consecutive releases and validating the results manually it is possible to gain insights in the strengths and weaknesses of the matching system and possibly tweak the system to produce better results.

The second case study will try to identify similar student projects. For this experiment a large set of submissions for several student assignments will be used. It is assumed that the submissions for an assignment are similar in some way. The goal of this experiment is to use ontology matching to group the submissions for each assignment.

4.1 Selecting an ontology matcher

There is a large collection of ontology matching techniques and tools available. So how to know which one to use? The *Ontology Alignment Evaluation Initiative* (OAEI)[3] is an international initiative that compares matching systems to determine the best matching strategies. They run a contest each year in which different systems compete against each other. Of the 21 systems that participated in the 2016 contest, *AgreementMakerLight* (AML) performed the best overall [39]. AML is a matching framework that has multiple matching techniques and can easily be extended with custom matching algorithms [40] [41]. After some investigation, AML seems like a good choice. It has a public codebase on GitHub and is actively maintained.

[2] https://commons.apache.org/proper/commons-collections/
[3] http://oaei.ontologymatching.org/

4.2 Defining the ontology

It is not feasible to capture all semantic information just from the source code of a project. However, well-written code will reveal a lot of semantic information just by the names of classes, methods and parameters alone [37]. Consider for example a class called `ShoppingCart` containing the methods `addItem` and `removeItem`. The names of the class and the methods give a lot of information about the intended use of this class.

To capture this information, the ontology used in this study is based on the ontology created by Ganapathy and Sagayaraj [42]. The ontology types `Project`, `Package`, `Class`, `Method` and `Parameter` are defined as subclasses of `Thing` and are declared as being disjoint with each other. These classes represent their Java source code counterparts. The disjoint axiom between the classes states that they cannot share an instance. To prevent confusion between ontology classes and software classes, ontology classes will be referred to as *types* in the remainder of this document.

Next, four object properties are defined: `isPartOfProject` with domain Package and Range Project, `isPartOfPackage` with domain Class and range Package, `isPartOfClass` with domain Method and range Class and finally `isPartOfMethod` with domain Parameter and range Method. The XML for this ontology can be found in appendix A. These relations are used to create relations between individuals from the corresponding domain and range.

Preferably, all packages, classes, methods, and parameters that are extracted from the source code should be added as an individual to the ontology. However, initial testing showed that AML cannot create matches between individuals. Therefore all packages, classes, methods, and parameters are created as separate types in the ontology and instead of being made a member of a type, they are declared as being a subtype of a type. As an ontology type name needs to be unique, identifiers are used as the name and the actual name is stored in the label annotation.

4.3 Extracting ontologies from source code

Extracting ontologies from source code has been done before. Ganapathy and Sagayaraj used a method to automatically extract ontologies in OWL format from JAVA classes [42]. Their method depends on *QDox*[4] to extract meta data from the source code and uses *Apache Jena* [5] to create and store the OWL definition. However QDox does not seem to be maintained

[4]https://github.com/paul-hammant/qdox
[5]https://jena.apache.org/

anymore so after some experiments with different source code extraction tools, *Spoon*[6] was chosen. Spoon creates an abstract syntax tree (AST) from a Java application that can be used to query or transform the source code. The AST can be processed and Apache Jena can be used to generate an ontology. Apache Jena is a Java library that provides an API to create ontologies. To inspect and verify the generated ontologies *Protg*[7] is used. Protg is a tool for creating and editing ontologies created by the University of Stanford.

Below is the ontology definition for the class `org.apache.commons.collectionsFastTreeMap` as created by the extractor tool. In this example `http://www.semanticweb.org/apache-collections-v1.0/#C14` is the unique identifier for this type and it is declared to be a subclass of type `class`. The label contains the class name `FastTreeMap` that will be used by AML and `comment` contains the qualified class name. The comment is not used by AML, but is added by the extractor as this can be used to compare the final alignment with the golden standard. Finally, another subclass is added which indicates that at least one of the types that has a `isPartOfPackage` relation with this type should be of type `http://www.semanticweb.org/apache-collections-v1.0/#P1`, which is the package the class is in.

```
<owl:Class rdf:about="http://www.semanticweb.org/apache-collections-v1.0/#C14">
    <rdfs:subClassOf rdf:resource="http://www.semanticweb.org/ontologies/#Class"/>
    <rdfs:label xml:lang="EN">FastTreeMap</rdfs:label>
    <rdfs:comment xml:lang="EN">org.apache.commons.collections.FastTreeMap</rdfs:comment>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:someValuesFrom rdf:resource="http://www.semanticweb.org/apache-collections-v1.0/#P1"/>
        <owl:onProperty>
<owl:ObjectProperty rdf:about="http://www.semanticweb.org/ontologies/#isPartOfPackage"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

The extraction process only extracts public methods from a class. The assumption is that these are well named and clearly state what the purpose of that particular method is as these are meant to be used outside of the class context. Private and protected methods are hidden from the rest of the application and the naming of these methods only need to make sense in the context of their containing class.

[6] http://spoon.gforge.inria.fr/
[7] http://protege.stanford.edu/

5 AgreementMakerLight

The AML agreement creation process can be divided into 4 stages: ontology loading and data structure initialization, translation, matching, and selection. A schematic overview is given in figure 3.

1. In the first stage AML loads the source and target ontologies and generates the lexicon and relationship map data structures for each of the ontologies. The *lexicon* stores all lexical information for an ontology. It can be used to look up the names, label annotations, and synonyms for a class in an ontology. The *relationship map* stores the structural information for an ontology. It can be used to look up all related classes for a class in the ontology. These data structures can later be used during the matching stage.

2. In the translation stage, AML will look if there are multiple languages used in either two ontologies. If this is the case, AML will lookup translations for all words using the Microsoft Translation API [8]. In this study most texts are in English so the translation stage is ignored.

3. Then the actual matching is done by a set of matching algorithms, so-called *matchers*. Every matcher takes the source and target ontologies as input and produces an alignment as a set of *mappings*. A mapping contains a type from the source ontology, a type from the target ontology and a *similarity score*. This score is a number between 0 and 1 where 1 indicates that the two types are completely similar to each other and 0 indicates that the two types are completely different.

4. In the final stage selection algorithms remove undesired matches from the alignment.

5.1 Matchers

AML contains a framework that can run multiple matchers during the matching process. The order of these matchers is important as AML employs the matchers sequentially and under the assumption that every consecutive matcher has lesser precision than the previous one. Therefore, existing mappings can never be removed by matchers. Matchers are only allowed to improve similarity scores for mappings created by previously executed

[8] https://www.microsoft.com/en-us/translator/default.aspx

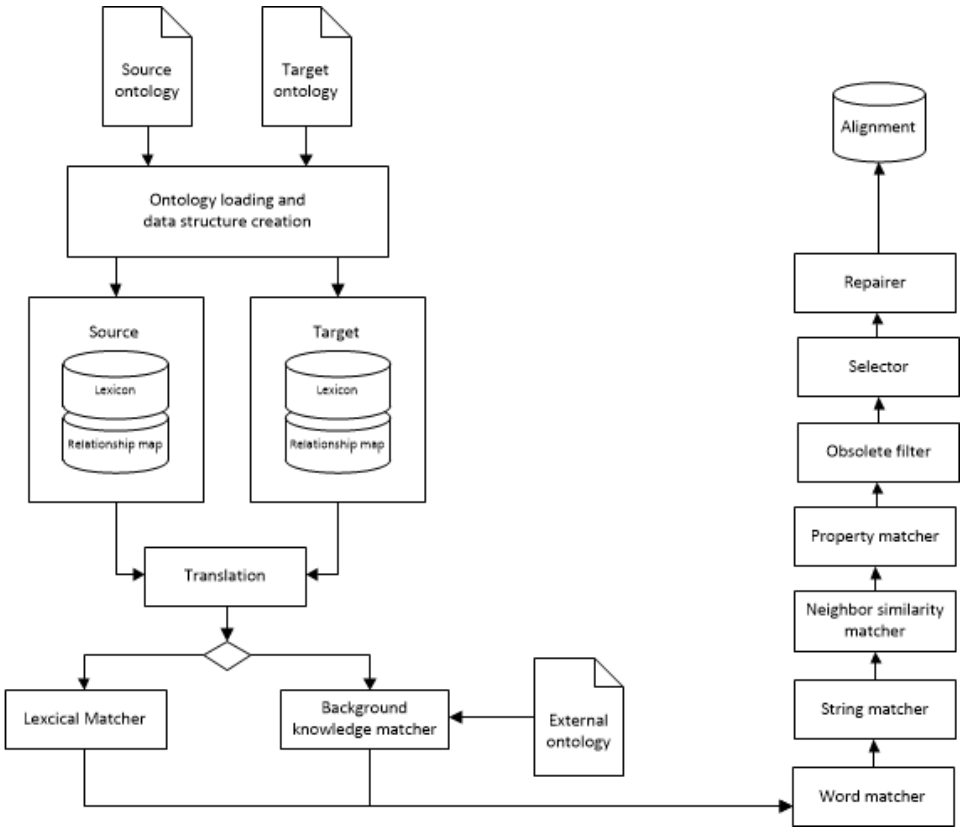

Figure 3: The AML alignment creation process

matchers. To make sure only valid mappings are added to the alignment, AML uses a similarity score threshold. Mappings that have a score below this threshold will not be added to the alignment.

Matchers can be divided into two groups: primary matchers and secondary matchers. *Primary matchers* are matchers that calculate similarities between all the types in the source ontology and all the types in the target ontologies. As this can be a very time-consuming operation, a primary matcher in AML should use the internal data structures to minimize its execution time. *Secondary matchers* only improve existing matches created by primary matchers. These secondary matching algorithms are often slow and do not scale well with large ontologies.

As the matchers are essential for this study, a good understanding of the underlying algorithms employed by these matchers is required. Unfortu-

nately, most information found in papers was either outdated or incomplete. Therefore, one of the contributions of this study is the reverse engineering and documentation of these matching algorithms.

The general matching algorithm is stated below. For the remainder of this document *source* and *target* are the input ontologies and A is the final alignment returned by the matching framework. An alignment is a set of *mappings* where a mapping is a 3-tuple containing a type from the source ontology, a type from the target ontology and a similarity score.

```
function match(source, target)
  A = ∅
  for matcher ∈ matchers
    if isPrimary(matcher)
      aux = matcher.match(source, target)
    else
      aux = matcher.rematch(A)

    for m in sortBySimilarity(aux)
      if ¬(m.sType ∈ (sType | (sType, tType, sim) ∈ A))
        ∨ m.tType ∈ (tType | (sType, tType, sim) ∈ getMappingsBySourceType(m.sType))
          if m.similarity > thresh
            A = A ∪ {mapping}
  return A
```

AML has the following built-in matchers. They are listed in the order of execution as can be seen in figure 3.

- *Lexical matcher*: a primary matcher that tests for literal name matches between two types.
- *Background knowledge matcher*: a primary matcher that uses external ontologies to find matches between the types in the source and target ontologies (see section 5.1.2 for more information about the use external ontologies). It also incorporates the lexical matcher internally so running the background knowledge matcher makes the use of the lexical matcher redundant. In this study, only the WordNet ontology is used as an external source.
- *Word matcher*: a primary matcher that measures word similarity of the words in the names or labels of types.
- *String matcher*: matches elements by using string similarity algorithms on the names of those elements. It supports the ISub, Levenstein, Jaro-Winkler, and Q-gram string similarity algorithms. This matcher can be configured as either a primary matcher or a secondary matcher.
- *Neighbor similarity matcher*: a structural matcher that uses existing matches between all the children or parents of a type to calculate a

similarity score. This matcher is unique as it uses the information of previous matchers to calculate a similarity score. This is a very slow algorithm and is only available as a secondary matcher.

- *Property matcher*: a matcher that attempts to match properties between two ontologies. As property matches are not important in this study, this matcher will be ignored.

5.1.1 Lexical matcher

The lexical matcher is the most basic matcher in AML and matches the exact name of a type with the name of another type. The algorithm of the lexical matcher is described in [41] and is given below.

```
function lexicalMatcher(source, target)
  aux = ∅
  for name ∈ Lexicon
    for (sType,tType) ∈ getTypes(source, name) × getTypes(target, name)
      similarity = weight(name, sType) ∗ weight(name, tType)
      aux = aux ∪ {(sType, tType, similarity)}
  return aux
```

The lexical matcher uses the lexicon to retrieve all the names in the source and target ontologies and create a mapping between all types with the same name. AML allows types to have multiple names and assigns a different weight to these names. For example the type's identifier (when it is not a number or a number with one character prefix) has a weight of 1 and a type's label has a weight of 0.95 .

5.1.2 Background knowledge matcher

This matcher uses one or multiple external ontologies to create an alignment between the source and target ontologies. To do so, it uses the *mediating matcher* as well as the lexical matcher.

The mediating matcher is used to calculate an alignment between the source ontology and an external ontology and the target ontology and an external ontology. It first computes intermediate alignments between the source and target ontologies and the external ontologies. Then it cross searches these intermediate alignments. The mediating matcher algorithm is as follows where *med* represents a mediating ontology.

```
function mediatingMatcher(source, target, med)
  aux = ∅
  for (sMapping,tMapping) ∈ lexicalMatcher(source, med) × lexicalMatcher(target, med)
    if sMapping.targetType == tMapping.targetType
      similarity = min(sMapping.similarity, tMapping.similarity)
      aux = aux ∪ {(sMapping.sourceType, tMapping.sourceType,similarity)}
  return aux
```

The background knowledge matcher uses the mediating matcher to calculate an alignment for each external ontology and combines these results. The *gain* function calculates the fraction of new mappings in an alignment compared to the total number of mappings in a base alignment.

```
function backgroundKnowledgeMatcher(source, target)
  aux = lexicalMatcher(source, target)
  medAlignments = ∅
  for med ∈ mediatingOntologies
    auxMed = mediatingMatcher(source, target, med)
      if gain(auxMed, aux) > threshold
        medAlignments = medAlignments ∪ auxMed

  for medA ∈ orderByGain(medAlignments)
    if gain(medA, aux) > threshold
       aux = aux ∪ medA
  return aux
```

5.1.3 Word matcher

This matcher is a word-based string similarity matcher that calculates the similarity between two types by using the words in their names. It uses the lexicon data structure which contains a list of all words used in each ontology and also the *frequency* and *evidence content* (EC) for each word [43]. The frequency of a word is the number of type names that contain the word. The EC is calculated with the formula below. Where *maxFrequency* is the maximum frequency of all the words in the ontology.

$$EC_{(w)} = \log\left(\frac{\text{freq(w)}}{\text{maxFrequency}}\right)$$

The word matcher can use different strategies to calculate a similarity. These strategies are:

- CLASS
- NAME
- AVERAGE (average of CLASS and NAME)

- MAXIMUM (maximum of CLASS and NAME)
- MINIMUM (minimum of CLASS and NAME)

The CLASS strategy calculates similarities based on the EC of the types while the NAME strategy calculates similarities based on the EC of the words used in the type's name(s). Regardless of which strategy is chosen, the *class similarity* scores are calculated first for all the words and their corresponding types. If the word matcher is set to use the CLASS strategy, the word matcher is done and the produced alignment is appended to the main alignment. If another strategy is chosen, the *name similarity* for each mapping in the produced alignment is calculated. The *getNames* function returns the name, the label and the synonyms for a type and the *classEC* function calculates the EC value using all the words in a type's name.

```
function wordMatcher(source, target)
  aux = ∅
  for w ∈ getwords(source)
    for (sType, tType) ∈ getTypesContainingWord(source, w) × getTypesContainingWord(target, w)
      weightS = wordEC(source, w) · wordWeight(w sType)
      weightT = wordEC(target, w) · wordWeight(w, tType)
      similarity = SQRT(weightS · weightT)

      if containsMapping(aux, sType, tType)
        m = getMapping(aux, sType, tType)
        m.similarity = m.similarity + similarity
      else
        aux = aux ∪ {(sType, tType, similarity)}

  for m ∈ aux
    s = classEC(m.sourceType) + classEC(m.targetType) − m.similarity
    m.similarity = m.similarity / s

  if ¬(strategy == CLASS)
    for m ∈ aux
      nameSim = calculateNameSim(m, source, target)
      if strategy == NAME
        m.similarity = nameSim
      else if strategy == AVERAGE
        m.similarity = avg(m.similarity, nameSim)
      else if strategy == MAXIMUM
        m.similarity = max(m.similarity, nameSim)
      else if strategy == MINIMUM
        m.similarity = min(m.similarity, nameSim)
  return aux
```

```
function calculateNameSim(m, source, target)
  nameSim = 0
  for (sName,tName) ∈ getNames(m.sourceType) × getNames(targetType)
    sWeight = getNameWeight(sName, m.sourceType)
    tWeight = getNameWeight(tName, m.targetType)
    sim = sWeight · tWeight · nameSimilarity(sName, tName)
    if sim > nameSim
      nameSim = sim
  return nameSim


function nameSimilarity(sName, tName)
  sharedNamesScore = 0
  maxNameScore = getNameEC(source, sName) + getNameEC(target, tName)
  for word ∈ getWords(sName)
    sharedNamesScore += SQRT(getWordEC(source, word) · getWordEC(target, word))
  return sharedNamesScore / (maxNameScore − sharedNamesScore)
```

5.1.4 String matcher

The string matcher uses string distance algorithms to calculate a similarity between two types. AML can use the following string distance functions by default:

- ISub
- Levenstein
- Jaro-Winkler
- Q-Gram

The algorithm for this matcher is listed below. The function *stringSimilarity* executes any of the string distance functions listed above.

```
function stringMatcher(source, target)
  aux = ∅
  for (sType, tType) ∈ source × target
    s = ∅
    for (sName,tName) ∈ getNames(sType) × getNames(tType)
      sim = stringSimilarity(sName, tName)
      s = s ∪ {sim · getWeight(sName, sType) · getWeight(tName, tType)) }
    if(MAX(s) > threshold)
      aux = aux ∪ {(sType, tType, MAX(s))}
  return aux
```

AML has a build in correction factor for the string matcher to make it score lower than the word matcher. This correction factor is optimized for the ISub algorithm and is hard-coded set to 0.8.

5.1.5 Neighbor similarity matcher

This is a unique matcher as it uses the output of the previous matchers to calculate the similarity between two types. It uses the relationship map to look up the parent and child types (depending on the strategy) of a type and compares these with the parent and child nodes of another type. This matcher is only implemented as a secondary matcher and uses a previously calculated alignment as input. Like the word matcher, this matcher has multiple strategies:

- ANCESTORS
- DESCENDANTS
- AVERAGE (average of ancestors and descendants)
- MAXIMUM (maximum of ancestors and descendants)
- MINIMUM (minimum of ancestors and descendants)

The main algorithm is listed below. Let *inputAlignment* represent the alignment that is calculated by one or more of the previous matchers.

```
function neighborSimilarityMatcher(inputAlignment)
  aux = ∅
  for m ∈ inputAlignment

  for sourceDescendant ∈ getDirectDescendants(m.sourceType)
    for targetDescendant ∈ getDirectDescendants(m.targetType)
      sim = calculateSim(sourceDescendant, targetDescendant)
      aux = aux ∪ {(sourceDescendant, targetDescendant, sim)}

  for each sourcAncestor in getDirectAncestors(m.sourceType)
    for each targetAncestor in getDirectAncestors(m.targetType)
      sim = calculateSim(sourceAncestor, targetAncestor)
      aux = aux ∪ {(sourceAncestor, targetAncestor, sim)}

  return aux
```

The *calculateSim* function uses the *calculateAncestorsSim* when the DESCENDANTS strategy is selected, the *calculateDescendantsSim* function when the ANCESTORS strategy is selected and both function for the other strategies. The algorithm for the calculateAncestorsSim function is listed below. The calculateDescendantsSim function is similar to the calculateAncestorsSim function, with the only exception that *getAncestors* is replaced by *getDescendants*.

```
function calculateAncestorsSim(sType, tType)
  sim = 0
  maxSim = 0

  for sParent ∈ getAncestors(sType)
    sd = getDistance(sParent, sType)
    maxSim += 0.5 / sd
    for tParent ∈ getAncestors(tType)
      td = getDistance(tParent, tType)
      sim += inputAlignment.getSim(sParent, tParent) / SQRT(sd · td)

  for tParent ∈ getDirectAncestors(tType)
    maxSim += 0.5 / getDistance(tParent, tType)

  return sim / maxSim
```

5.2 Selection algorithms

Selection algorithms remove undesired matches from the alignment. AML has three selection algorithms:

- *Obsolete filter*: this filter can remove mappings that are marked as obsolete by either the user or a reference ontology. In this study neither of those are available, so this filter will be ignored.
- *Selector*: a filtering algorithm based on cardinality. It can be configured to run in different modes. In strict mode only the mapping with the highest similarity score is selected for a type from the source ontology. In permissive mode multiple mappings are allowed if they have an equal similarity score. And in hybrid mode it allows a type to be mapped to a maximum of two other classes as long as these individual mappings have a similarity score of more than 0.75.
- *Repairer*: this algorithm removes mappings between types that are marked as disjoint in an ontology. In our ontology that could be mappings between classes and methods for example.

5.3 Profiler

As the output of AML is only the calculated alignment it is impossible to see the contributions of each matcher to the final result. To gain some more insight into the contribution of each matcher to the final score we created a profiler and integrated it into AML. The profiler records every match that is added to the alignment, even if that match is lower than an existing match and would otherwise be discarded by AML. It records the matcher that is

responsible for the match and the similarity score of the match. Besides the matches, it also records the total number of classes and relationships in the ontologies, the run time of each individual matcher, the number of mappings removed by each of the selection algorithms and total run time of the alignment creation process. Recording the contribution of each matcher per mapping is very time consuming and memory insensitive. Therefore this feature will only be disabled by default and only enabled when required.

5.4 Configuration

AML can be run in two modes: *automatic* and *manual* mode. In automatic mode, all settings are configured automatically using a build in configurator that behaves differently depending on the size of the ontology. In manual mode, a configuration file can be used to configure AML. The table below lists all AML configuration settings that are relevant for this study.

Some preliminary tests revealed that the WordNet ontology is removed by the automatic configurator for ontologies with more than 500 types due to performance reasons. The automatic configurator also disables the structural matcher for ontologies larger than 500 classes.

5.5 Simple demonstration

To demonstrate the performance of each matcher a small test is created. Two projects are created: demo1 and demo2. Below is the JAVA code for both projects and the golden standard is given in table 2.

```
package demo1;

public class CanvasDrawingContainer {}
public class Circle{}
public class Rectangle {}
public class Star {}


package demo2;

public class BetterCanvasDrawingContainer {}
public class Circle {}
public class Rectangl {}
public class Asterisk {}
```

As class `demo1.CanvasDrawingContainer` and `demo2.BetterCanvasDrawingContainer` contain 3 and 4 words respectively and only have one word difference it is a good candidate to get matched by the word matcher. The mapping from `demo1.Circle` to `demo2.Circle` is trivial and should be made by the lexical matcher. The string matcher

setting	possible values	description
`bk_sources`	`All`, `none` or a comma-separated list of file names"	Enables or disables the background knowledge matcher and contains a list of background sources that can be used by the matcher. For this study only the options `none` and `WordNet` are relevant.
`word_matcher`	`by_class`, `by_name`, `average`, `maxmimum`, `minimum`, `maximum`, `auto`, `none`	Enables or disables the word matcher and selects the word matching strategy to use. The `auto` option is the same as the `average` option.
`string_matcher`	`none`, `global`, `local`, `auto`	Enables or disables the string matcher and selects if it should run in primary mode (global) or secondary mode (local). The `auto` option uses the string matcher in primary mode for classes with less than 500 classes, else it will use the matcher in secondary mode.
`string_measure`	`ISub`, `Levenstein`, `Jaro-Winkler`, `Q-gram`	The string similarity algorithm that is used by the string matcher.
`struct_matcher`	`ancestors`, `descendants`, `average`, `maximum`, `minimum`, `none`, `auto`	Enables or disables the structural matcher and selects the matching strategy. The `auto` option is the same as the `descendants` option.
`selection_type`	`strict`, `permissive`, `hybrid`, `auto`, `none`	Sets the strategy for the selector. The `auto` option uses the `strict` option for ontologies with less than 500 classes, the `permissive` option for ontologies with more than or exactly 500 classes and less than 5000 and the `hybrid` option for all other cases.

Table 1: Configuration options for AML in manual mode

Demo 1	Demo 2
demo1.CanvasDrawingContainer	demo2.BetterCanvasDrawingContainer
demo1.Circle	demo2.Circle
demo1.Rectangle	demo2.Rectangl
demo1.Star	demo2.Asterisk

Table 2: Class mappings for demo projects

Setting	Value
bk_sources	WordNet
word_matcher	auto
string_matcher	auto
string_measure	ISub
struct_matcher	auto
selection_type	auto
repair_alignment	true

Table 3: Manual configuration for demo project alignment creation

Source	Target	Lexical	Bckgrnd	Word	String	Struct
demo1.Canvas-DrawingContainer	demo2.Better-CanvasDrawing-Container			0.7	0.62	0
demo1.Circle	demo2.Circle		0.88	0.94	0.71	0
demo1.Rectangle	demo2.Rectangl				0.68	0
demo1.Star	demo2.Asterisk		0.78			0

Table 4: Similarity score per mapping per matcher for demo project

should be able to match `demo1.Rectangle` and `demo2.Rectangl` as they only differ one letter (in real life, this could for example be a corrected typo). And finally, the background knowledge matcher with the WordNet ontology should be able to match `demo1.Star` and `demo2.Asterisk` as they are listed as synonyms in WordNet.

The extractor tool is used to create the ontologies for the two projects and AML is used in manual mode with the configuration values listed in table 3. The produced alignment contained only correct mappings. Table 4 contains all the class mappings in the alignment and the similarity score calculated for that mapping by every individual matcher. The lexical matcher is not separately visible in the profiler output as it is integrated into the background knowledge matcher. The string similarity score for the Circle mapping is lower than initially would have been expected, but this is caused by the corrections for the labels (0.95) and the correction for the string matcher itself (0.8).

5.6 Neighbor similarity matcher in primary mode

The neighbor similarity matcher has a lot of potential when searching for similar software. As the ontology used in this experiment is hierarchical, classes could be found based on the similarity of their methods. However, the neighbor similarity matcher can only be used in secondary mode e.g. it can only strengthen matches that are already found by other matchers. To use the neighbor similarity matcher in primary mode, we will change the main algorithm given in 5.1.5.

```
function neighborSimilarityMatcherPrimary(source, target)
  aux = ∅
  for (sType, tType) ∈ source × target
    aux = aux ∪ {(sType, tType, calculateSim(sType, tType))}
  return aux
```

In the `calculateDescendantsSim` and `calculateAnsectorsSim` function some changes were required because the original implementations of the `getAncestors` and `getDescendants` functions did not recognize the conditional `isSubTypeOf` relation in the ontology used in this study. Another small change is made so this matcher will be disabled if a type only has one child or parent as one sample is not enough to establish that two types are equal. To use this matcher in primary mode, a new configuration setting is created. This setting is `struct_matcher_mode` and its values are either `primary` or `secondary` where `secondary` is the default value.

To test the matcher in primary mode, two more classes are added to the demo projects. These classes have totally different names but have largely the same methods. The configuration is the same as table 1 where the `struct_matcher_mode` is set to `primary`. The result listed in table 5 show that AML is now able to match classes that have completely different names, but similar methods.

```
package demo1;

public class Triangle {
  public int getSides() { return 3; }
  public String getName() { return "demo.Triangle"; }
}


package demo2;

public class ShapeWith3Sides {
  public int getSides() { return 3; }
  public String getName() { return "demo.ShapeWith3Sides"; }
  public boolean isTriangle() { return false; }
}
```

Source	Target	Lexical	Bckgrnd	Word	String	Struct
demo1.Canvas-DrawingContainer	demo2.Better-CanvasDrawing-Container			0.7	0.62	0
demo1.Circle	demo2.Circle		0.88	0.94	0.71	0
demo1.Rectangle	demo2.Rectangl				0.68	0
demo1.Star	demo2.Asterisk		0.78			0
demo1.Triangle	demo2.ShapeWith3Sides					0.75

Table 5: Similarity score per class per matcher for demo project with neighbor similarity matcher user in primary mode

5.7 Threshold

AML uses a *threshold* to prevent mappings with low scores from being added to the output alignment. The threshold filter is applied to the output of each individual matcher and removes all mappings that have a similarity score below that threshold from the output. As this threshold value is of significant importance, AML was changed to make the threshold configurable. A new configuration `threshold` is created which can range from 0 (no threshold applied) to 1 (only exact matches are allowed) and has a default value of 0.6.

6 Case study - Apache commons collection

In the first case study we will try to match classes in consecutive versions of the Apache Commons Collections [9]. The goal of the Apache Commons Collection project is to build upon the Java Collections Framework by providing new interfaces, implementations, and utilities. This open source software initiative has a large codebase, 16 releases and 4 major version updates with commits dating back to July 2007. It also has an excellent issue tracking system that can provide background information on the changes in different versions. By using the matcher on consecutive releases and validating the results manually it is possible to gain insights into the strengths and weaknesses of the matching system and possibly tweak the system to produce better results.

Only the latest stable versions of each release are used in this phase. These versions are 1.0, 2.1.1, 3.3 and 4.1. Any unit tests in the packages will be excluded in this experiment as these classes are not designed to be used outside of the project itself, and do not have sensible names. Table 6 contains information about the ontologies used in this study.

All experiments were run on a virtual private server (VPS) with 6 Intel Xeon CPU E5-2630L v2 CPUs running at 2.40GHz and 16GB RAM. Unfortunately, during the tests it was found that the VPS did not give a constant performance. A test running multiple times with the same configuration could end up with different run times. So the run times listed in this study should not be taken literally and are purely listed here to get a basic indication of the performance differences between the various algorithms.

version	packages	classes	methods	parameters	extraction duration (sec)
1	1	23	374	338	6
2.1.1	2	62	839	606	12
3.3	12	250	3909	2846	152
4.1	18	267	4279	3260	203

Table 6: Apache Commons Collections versions ontology information

6.1 Golden standard

A golden standard is the perfect mapping from the classes in one version of the Apache Commons Collections project to another. When comparing

[9] https://commons.apache.org/proper/commons-collections/

From version	To version	Number of mappings
1.0	2.1.1	29
2.1.1	3.3	69
3.3	4.1	216

Table 7: Number of mappings in the golden standards for the Apache Commons Collection versions

a generated alignment with a golden standard, the precision, recall and F-measure score can be calculated. For the experiment with the Apache Commons Collections, three golden standards are required: a mapping of version 1 to version 2.1.1, a mapping of version 2.1.1 to 3.3 and a mapping from version 3.3 to 4.1.

The creation of these mapping is a manual process. For each class in one version of the project, the equivalent class in the next version is selected. The Apache Commons Collections project has detailed release documentation for every version and all source code changes are publicly visible in the Git repository. With this information, every class can be tracked from one version to another. This process was done manually by the author and was reviewed by an experienced software engineer. Table 7 lists the amount of mappings per golden standard.

6.2 Configuring AML

AML comes with a lot of configuration options that need to be tuned to optimize its performance for the experiments. To find out what the optimum configuration is, several configurations will be tested to create alignments between the ontologies of different versions of the Apache Commons Collections package.

6.2.1 Setting a baseline with AML in automatic mode

To create a baseline, AML is run in automatic mode. To make sure that the profiler only impacts the run time and not the results, AML is run in automatic mode twice: once with the profiler disabled and once with the profiler enabled. The results are stated in table 8 and the profiler results are listed in tables 9 and 10.

The scores are already very good: the alignment from version 1 to 2.1.1 is perfect, the alignment from 2.1.1 to 3.3 missed one out of 70 mappings and

Source	**Target**	**Precision**	**Recall**	F_1	**Runtime**	**Runtime with profiler**
1.0	2.1.1	1.0	1.0	1.0	33	36
2.1.1	3.3	1.0	0.99	0.99	375	458
3.3	4.1	1.0	0.95	0.97	18815	21507

Table 8: results of Apache Commons Collections matching on auto mode

	v1.0 - v2.1.1			v2.1.1 - v3.3			v3.3 - v4.1		
	added	improved	runtime	added	improved	runtime	added	improved	runtime
Lexical	18752 (29)	0	0.6	126405 (68)	0	14	1004864 (207)	0	1187
Bckgrnd	0	0	14	0	0	15	0	0	13
Word	0	80 (17)	0.5	0	104 (29)	16	0	167 (79)	1.240
String	1 (0)	80 (17)	7	3 (0)	104 (29)	198	10 (0)	160 (79)	2652
Struct	0	0	0.7	-	-	-	-	-	-

Table 9: Contribution of each matcher for Apache Commons Collections matching on auto mode with profiler. Matches between classes are listed between parenthesis.

	v1.0 - v2.1.1		v2.1.1 - v3.3		v3.3 - v4.1	
	removed	runtime	removed	runtime	removed	Runtime
Selector	340 (0)	0.6	3979 (0)	10	20978 (2)	651
Repairer	240 (0)	2	4007 (0)	149	13172 (0)	29303

Table 10: Contribution of each selector for Apache Commons Collections matching on auto mode with profiler. Matches between classes are listed between parenthesis.

the alignment from 3.3 to 4.1 missed 11 out of 217 mappings. No incorrect mappings were made.

The profiler data reveals that the structural matcher is only used when creating the alignment between versions 1 and 2.1.1. Studying the code learns that the automatic configurator disables the structural matcher for ontologies that contain more than 500 types. Another thing to note is that the repairer does not scale well for larger ontologies. For the creation of the alignment between version 3.3 to 4.1 it consumed about 71% of the total runtime while only removing mappings that have no impact on the final result.

6.2.2 Configuring AML in manual mode

Now that a baseline is set AML is switched to manual mode. To get the optimal usage out of AML in manual mode, all different variants of the settings should be explored. This would however, result in thousands of possible combinations and is not feasible. Therefore one setting at a time will be changed to observe what the impact is for that setting on the overall score. In the end, the best value for each individual setting will be selected and combined into a final configuration.

For the first manual run, the same configuration values as the auto configurator generates are used. The only difference is the setting for `bk_sources` as the automatic matcher runs both the lexical matcher and the background knowledge matcher and the manual mode can only run one of them. As the background knowledge matcher in automatic mode removes the WordNet ontology it does basically nothing. So the background knowledge matcher is disabled in favor of the lexical matcher. The configuration values are listed in table 11, the results in table 12 and the profiler results in tables 13 and 14.

It is interesting to note that the structural matcher suddenly improves existing matches while this was not the case when using AML in automatic mode. This can be explained by the fact that the automatic configuration changes the threshold for the matchers based on the ontology size, while in manual mode a fixed threshold is selected.

Again the repair selector did not contribute anything to the final result. This selector removes matches between ontology types that are disjoint. For example a match between a method and a parameter. We already created a filter for the output to show only class mappings, so this selector will have no impact on our final result. Therefore, this selector will be disabled by setting the `repair_alignment` setting to `false` and will be ignored in the

Setting	Value
bk_sources	none
word_matcher	auto
string_matcher	auto
string_measure	ISub
struct_matcher	auto
selection_type	auto
repair_alignment	true
struct_matcher_mode	secondary
threshold	0.6

Table 11: Manual configuration for demo project alignment creation

Source	Target	Precision	Recall	F_1	Runtime (sec)
1.0	2.1.1	1.0	1.0	1.0	17
2.1.1	3.3	1.0	0.99	0.99	466
3.3	4.1	1.0	0.95	0.97	21925

Table 12: Results of Apache Commons Collections matching on manual mode with auto mode config values

	v1.0 - v2.1.1			v2.1.1 - v3.3			v3.3 - v4.1		
	added	improved	runtime	added	improved	runtime	added	improved	runtime
Lexical	18752 (29)	0	0.9	126405 (68)	0	15	1004864 (207)	0	1275
Bckgrnd	-	-	-	-	-	-	-	-	-
Word	0	80 (17)	0.4	0	104 (29)	8	10 (0)	160 (79)	657
String	1 (0)	80 (17)	8	3 (0)	104 (29)	205	3 (0)	167 (79)	3369
Struct	0	83 (17)	0.5	0	109 (29)	22	0	177 (79)	2140

Table 13: Contribution of each matcher for Apache Commons Collections matching on manual mode with auto mode config values. Matches between classes are listed between parenthesis.

	v1.0 - v2.1.1		v2.1.1 - v3.3		v3.3 - v4.1	
	removed	runtime	removed	runtime	removed	Runtime
Selector	340 (0)	0.7	3979 (0	9	20978 (2)	705
Repairer	240 (0)	4	4007 (0)	224	13172 (0)	15.913

Table 14: Contribution of each selector for Apache Commons Collections matching on manual mode with auto mode config values. Matches between classes are listed between parenthesis.

rest of the study.

In consecutive tests different values for the `bk_source` and `struct_matcher` (while running the structure matcher in secondary mode) had no effect on the score. Setting the `string_measure` to `Jaro-Winkler` decreased the score slightly and other values for the setting made no difference with regard to the baseline. All values for the `selection_type` setting gave the same result. Disabling the selection however, produced a slightly worse result as some false positives were no longer removed from the final alignment. For the `word_matcher` the `by_class` setting (and thus the `maximum` setting) increased the score slightly by finding one extra match in the alignment from version 3.1 to 4.4. The `threshold` setting was tested with the values 0.4 and 0.8. With the 0.8 value there was no difference when compared to the baseline. The 0.4 value result in more incorrect mappings.

Next, the neighbor similarity matcher was run in primary mode by changing `struct_matcher_mode` to `primary`. To test if this new matcher behaves correctly, the matcher was tasked to compare the 1.0 version of the project with a modified 1.0 version where all classes names were converted to a number. The matcher found some incorrect mappings, but overall performed very well.

As can be seen in the results in table 15, the `descendants` strategy, and consequently the `maximum` strategy had a negative impact on the F_1 score for the 3.3 to 4.1 mapping. The matcher was able to find one extra match, but also added 9 incorrect matches. Studying the incorrect matches revealed that these classes contained mostly the same methods. It also reveals that this type of matcher is not very good in a library where most classes have almost the same public interface. As is the case for the Apache Commons Collections library where almost all classes contain methods like `add`, `remove` and `contains`.

Strategy	**Source**	**Target**	**Precision**	**Recall**	F_1
ancestors	1.0	2.1.1	1.0	1.0	1.0
	2.1.1	3.3	1.0	0.99	0.99
	3.3	4.1	1.0	0.95	0.97
descendants	1.0	2.1.1	1.0	1.0	1.0
	2.1.1	3.3	0.99	0.99	0.99
	3.3	4.1	0.96	0.96	0.96
average	1.0	2.1.1	1.0	1.0	1.0
	2.1.1	3.3	1.0	0.99	0.99
	3.3	4.1	1.0	0.95	0.97
maximum	1.0	2.1.1	1.0	1.0	1.0
	2.1.1	3.3	0.99	0.99	0.99
	3.3	4.1	0.96	0.96	0.96
minimum	1.0	2.1.1	1.0	1.0	1.0
	2.1.1	3.3	1.0	0.99	0.99
	3.3	4.1	1.0	0.95	0.97

Table 15: Results of Apache Commons Collections matching on manual mode with structure matcher in primary mode

Setting	Value
bk_sources	WordNet
word_matcher	by_class
string_matcher	global
string_measure	ISub
struct_matcher	descendants
selection_type	auto
repair_alignment	false
struct_matcher_mode	secondary
threshold	0.6

Table 16: Final AML configuration for the Apache Commons Collection experiment

6.2.3 Final configuration

Although the background matcher did not improve the score, the example projects show there that it can lead to better results, as such it will be enabled. The `by_class` strategy scored best for the word matcher. The string matcher will be run in primary mode as this has no drawbacks except consuming more processor power. The best string matcher algorithm could not be determined with the current dataset, however, a study by Yufei Sun, Liangli Maa and Shuang Wang [44] concluded that ISub is the best string measure metric for ontology alignment. And although the demo study in section 5.6 looked promising, the structural matcher in primary mode did not contribute much to the Apache Commons Collection case study. In fact, it lowered the precision when using the `descendants` strategy by introducing a lot of false positives. Therefore this matcher will be run in secondary mode and will use the descendants strategy. The final configuration that will be used for this experiment is listed in table 16.

6.3 Results

The final result using the configuration from table 16 can be seen in table 17. The configuration changes increased the F_1 score for the alignment between version 3.3 and 4.1 slightly when compared to the baseline configuration. No false positive were found by AML. The mappings that AML was not able to find are listed in table 18 and 19.

Source	Target	Precision	Recall	F_1
1.0	2.1.1	1.0	1.0	1.0
2.1.1	3.3	1.0	0.99	0.99
3.3	4.1	1.0	0.96	0.98

Table 17: Results of Apache Commons Collections matching with the configuration listed in table 16

V2.1.1 class	V3.3 class
collections.MultiHashMap	collections.map.MultiValueMap

Table 18: Missing mappings from v2.1.1 to v3.3 alignment

V3.3 class	V4.1 class
collections.ProxyMap	collections4.map.AbstractMapDecorator
collections.DefaultMapBag	collections4.bag.AbstractMapBag
collections.buffer.PredicatedBuffer	collections4.queue.PredicatedQueue
collections.BufferUtils	collections4.QueueUtils
collections.MultiHashMap	collections4.map.MultiValueMap
collections.buffer.UnmodifiableBuffer	collections4.queue.UnmodifiableQueue
collections.buffer.AbstractBufferDecorator	collections4.queue.AbstractQueueDecorator
collections.buffer.TransformedBuffer	collections4.queue.TransformedQueue
collections.DoubleOrderedMap	collections4.bidimap.TreeBidiMap

Table 19: Missing mappings from v3.3 to v4.1 alignment

6.4 Conclusion

The overall result is very good. The vast majority of mappings were found and there were no false positives. The mappings that AML was not able to find are even for a human hard to identify. And some of these could only be established by using the bug tracking system of the Apache Commons Collection project.

Further investigation regarding the missing mapping listed in table 18 revealed that this mapping was not found because of two reasons. The first reason was that the most likely matcher to find this mapping, namely the word matcher, calculated a similarity score of 0.57 for this mapping. This is just below the threshold of 0.6. Another reason why this match was not found, it the fact that there was already a better mapping for collections.MultiHashMap. This is one of the rare cases where one class in the source version was matched to multiple classes in the target version. As AML only allows one match for a class in the source version, it would never be able to find this match. Investigation of the missing mappings listed in table 19 yielded similar findings.

It should be noted that a large number of the mappings are made by the lexical matcher (see table 7 and 9) and only a small portion of these mappings are made by other matchers.

6.5 Discussion

There are some threats to the validity of this experiment. First and most important is the golden standard as this is created by humans. Most mappings in the standard were unanimously agreed upon by the author and the reviewer. However, some of these mappings were changed or removed after some discussions between the author and the reviewer. It is entirely possible that when other persons are asked to create a golden standard the results would be different. Unfortunately, this is something that cannot be avoided as the creation of these mappings requires human interpretation.

The second threat is caused by the configuration used in this experiment. As can be seen in section 6.2, the configuration has an impact on the final result. Due to time constraints, not all configurations are tested so it is possible that there is a better configuration that would be able to make AML score even better.

Finally, the amount of details in the ontology used for this experiment influences the outcome of the result. One could, for example, add type information to the ontology and create a matcher that takes the output type

and parameter types into account when comparing a method. It might first try to map the types between the two ontologies and use that information together with the name of the method to determine if there is a match. This might improve the final result. Code comments might also be added to the ontology although it should be noted that there is no guarantee that comments are up to date or actually describe the piece of code they are attached to.

7 Case study - student assignments

The second case study will try to group similar student projects. For this experiment, a large set of submissions to several student assignments were made available by Utrecht University. Table 20 gives an overview of the assignments and the number of submissions available.

	total		selection	
	submissions	classes	submissions	classes
animatedquicksort	219	1347	50	374
mandelbrot	1261	1477	50	69
petersonshortcut	103	654	50	324
reversi	1127	1986	50	79
spanningtree	118	639	50	272
threadedmergesort	87	671	50	378
treeroamer	46	437	46	437

Table 20: Student assignments information

As all submissions for an assignment implement the same specifications we can say that all submissions for an assignment are semantically similar. The goal of this experiment is to use ontology matching to group the submissions for each assignment. To do so, every submission will be compared to all other submissions. Because of time constraints, it is not feasible to use all submissions for this experiment. Therefore 50 submissions are randomly picked from each assignment. For the treeroamer assignment only 46 submissions will be used.

The configuration listed in table 21 will be used for this experiment. This is in large part the same configuration used in the first case study with the only exception being that the structural matcher will now run in primary mode. The structural matcher in primary mode did not perform well in the previous case study due to the high amount of shared method names in the classes of the Apache Commons Collection project. Considering the by average low number of classes for a submission, it is very unlikely that the classes belonging to a single submission will have a large number of shared method names. Therefore there is a low probability that the structural matcher in primary mode will report a large number false positives for this case study.

Setting	Value
bk_sources	WordNet
word_matcher	by_class
string_matcher	global
string_measure	ISub
struct_matcher	descendants
selection_type	auto
repair_alignment	false
struct_matcher_mode	primary
threshold	0.6

Table 21: Final AML configuration

7.1 Grouping the results

After AML is used to compare all the submissions the result needs to be grouped. To do so, a clustering algorithm will be used. This algorithm should meet the following criteria:

1. The algorithm is able to handle precalculated distances.
2. The algorithm is able to handle asymmetric data. As the distance from source to target can be different from the distance from target to source.
3. The algorithm is able to handle non-euclidean distances

The *DCSCAN* clustering algorithm meets these criteria and is well suited for this experiment. This algorithm tries to find core samples of high density and expands the clusters from them. Points that cannot be connected to a cluster are identified as noise. There are two parameters to influence the behavior: `eps` and `minSamples`. The `eps` parameter is the maximum *distance* between two samples for them to be considered to be in the same neighborhood. `minSamples` is the number of samples in a neighborhood for a point to be considered as a core point. The clustering package from *scikit-learn* [10] is used. The input for the clustering algorithm will be a list of three-tuples containing the source submission, the target submission and the distance between these submissions. The distance is a value between 0 and 1. A distance of 0 indicates the submissions are completely similar

[10] http://scikit-learn.org/stable/modules/clustering.htm

assignment	noise	0	1	2
animatedquicksort	34	16		
mandelbrot	17		33	
petersonshortcut	50			
reversi	9			41
spanningtree	50			
threadedmergesort	50			
treeroamer	46			

Table 22: Clustering results for the student assignments case study. `eps` = 0.05 and `minSamples` = 5

and a distance of 1 indicates these submissions are completely different. The formula to calculate the distance between two ontologies is listed below. Here *alignment* is the produced alignment between a *source* and target *ontology* which are extracted from two submissions.

$$distance = 1 - (\sum_{m \in alignment} m.simmilarity)/classcount(source)$$

7.2 Results

Again, the experiment was run on a virtual private server (VPS) with 6 Intel Xeon CPU E5-2630L v2 CPUs running at 2.40GHz and 16GB RAM. It took 79 hours and 26 minutes to create the 119716 alignments. From this set, 50 alignments were randomly selected and verified by comparing the respective source and target ontologies manually to confirm the validity of the creation process.

Tables 22, 23 and 24 show the final results with three different cluster configurations. The numbers in the top line of the table represent the different clusters. Table 22 uses the most strict configuration requiring 5 samples for a cluster core and a maximum distance of only 0.05 between two points. In table 23 the results are grouped in a way that a cluster core requires only 4 samples and the maximum distances between two points can be 0.2 and finally table 24 shows a configuration that requires only 3 samples to form a cluster core and the maximum distance is set to 0.4.

assignment	noise	0	1	2
animatedquicksort	29	18	3	
mandelbrot	1		49	
petersonshortcut	4		1	45
reversi	9		41	
spanningtree	15		1	34
threadedmergesort	42		7	1
treeroamer	35		11	

Table 23: Clustering results for the student assignments case study. `eps` = 0.2 and `minSamples` = 4

assignment	noise	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
animatedquicksort	15	31	4													
mandelbrot			50													
petersonshortcut	4		1	3	3	7	6	2	1	23						
reversi	2		48													
spanningtree	12		1		4	1	1	1		28	2					
threadedmergesort	25		8							1		10	2	4		
treeroamer	27		12												4	3

Table 24: Clustering results for the student assignments case study. `eps` = 0.4 and `minSamples` = 3

7.3 Conclusion

It is clear from table 22 that ontology matching performed well for the mandelbrot (33 out of 50) and reversi (41 out of 50) assignments and reasonably well for the animatedquicksort (16 out of 50) assignment as they were grouped together in separate clusters. When the cluster algorithm configuration is flexed a bit in table 23 it can be noted that the spanningtree (34 of 50 results) and petersonshortcut (45 out of 50 results) assignments were reasonably well grouped although they were both grouped in the same cluster. This could either mean that AML has identified a false positive, or that there actually is some similarity between these two assignments. It is likely to assume that there is a sweet spot for the eps setting that separates these assignments into different clusters. The overall grouping for the threadedmergesort and treeroamer assignments was very poor. Table 24 shows that if a more loose configuration is used for the clustering algorithm, some of the submissions are grouped but they are also put in the same clusters as submissions from other assignments.

It is interesting to note from table 20 that the reversi and mandelbrot assignments have the least amount of classes per submission: in average 1.7 and 1.2 classes per submission for the reversi and mandelbrot assignment respectively. Examining the samples it was found that almost all samples had a class that extended `Applet` and implemented the `ActionListener` and `MouseListener`. The small number of classes per submission and the relatively large amount of methods shared through the base class and interfaces explain why similar submissions could be identified by AML. It also explains why the submissions of both assignments are grouped in the same cluster in tables 23 and 24.

The fact that the animatedquicksort submissions cluster reasonably well can partially be explained by the fact that these submissions also use the `Applet` base class and one or more of the `ActionListener` and `MouseListener` interfaces. The reason that the animatedquicksort submissions do not cluster with the reversi and mandelbrot assignments is most likely because the interfaces and base classes make up a much smaller amount of the total number of classes in these submissions. Also, the fact that the samples use a lot of sorting terminology can help AML to differentiate these submissions from the rest.

Investigating the petersonshortcut and spanningtree submissions revealed that almost all submissions for these assignments had a main class called `GP2` or `GPD2`. Upon investigation this turned out to be an assignment requirement. All submissions from both assignments involved a client-server

and multi-threading implementation. The terminology used for these patterns can explain why both assignments were grouped in the same cluster. It should be noted however that although these are false positives when looked at the submission level, they can be considered true positives when looked at the class level.

The explanation for why most submissions end up in the same cluster is most likely due to the fact that interfaces and base classes make classes look very similar to AML. This way, submissions with a low class count can easily lead to false positives. Also the usage of generic class and method names like `main` and `controller` can easily lead to false positives.

Depending on the cluster configuration used it can be said that two assignments were clustered very well and that three other assignments were clustered reasonably well. It should be noted however that this experiment reveals that the matching performed very well at a class level, but that this effect does not automatically translate to good matches on a project level.

7.4 Discussion

The major threat to the validity for this experiment is that there is no golden standard. This implies that there is no way to make sure the alignments created by AML are actually valid. Consequently, the assumption that all submissions for an assignment are similar to each other cannot be verified without looking at all the submissions manually. It might be the case that some students misread the assignment or for some other reason submitted something different than that what was asked in the assignment. However, this chance is relatively low.

The clustering algorithm and the configuration for this algorithm used to present the results can be considered a threat. As can be seen in tables 22, 23 and 24, different cluster configurations can lead to very different representations and thus interpretations of the results.

Finally, it is possible that the AML configuration and the amount of information in the ontologies had a significant impact on the result. It is possible that there is a better AML configuration or that ontologies with more information will yield better results.

8 Conclusion and future work

8.1 Conclusion

The objective of this thesis was to decide whether ontology matching techniques could be employed to identify semantically similar software. To do so tooling was created to extract semantic information from a JAVA project and store this in an ontology. The actual ontology matching is done by AML. AML performs very well on comparing medical ontologies but was never used to compare ontologies extracted from software projects. Some modifications to the AML infrastructure and matchers were required and multiple test runs were done to determine the best configuration for the case studies.

Two case studies were conducted to validate if AML can be used to find similar software. For the first case study, ontologies were extracted from four major release versions of the Apache Commons Collections projects. AML was used to create three alignments between the classes of those versions. The available project documentation and bug tracker was used to create golden standards which were used to validate the alignment. The results of this case study were very promising. The alignments contained no false positives and contained almost all available class mappings.

The second case study used a data set of 346 submissions for seven student assignments from programming courses given at Utrecht University. The goal of this case study was to group the submissions per assignment. Alignments were made between the student submissions and a clustering algorithm was used to group the results. Five of seven assignments were grouped very well although there were a lot of false positives between the mandelbrot and reversi assignments and the petersonshortcut and spanningtree assignments. In the case of mandelbrot and reversi the false positives were introduced by the combination of a low class count per submission and the usage of interfaces and base classes to render a UI. In the case of the petersonshortcut and spanningtree the false positives were introduced by the use of similar design patterns. These false positives are however true positives when considered at class level.

Considering RQ 1.1.1, we can state that ontology matching showed great promise as a technique to find semantically similar software on a class level. Considering that the matching software itself was not tailored towards software comparison it is expected that even better results might be accomplished.

8.2 Future work

There are several aspects that could be improved by future research. First and foremost: more research is required to answer RQ 1. As this study concludes that it is possible to find semantically similar software, future research should focus on the challenges of automatically locating and converting test cases so they can actually be reused. To locate test cases that cover a unit of code, it could perhaps be possible to exploit the fact that many projects in this day and age use *continuous integration* connected to a *version control system*. This often implies that information about the location of test cases and execution information is present within a project repository. Advancements in the field of *automated software repair* might be used to convert test cases from one software project to another.

Beside these major challenges, there is also future work that could be done to improve the identification of semantically similar software. The information stored in the ontologies used in this research heavily influenced the results. Experiments with ontologies containing more or different information could perhaps reveal better ways to utilize the ontology matching software.

Another very interesting topic for future research would be to investigate if it is worthwhile to use semantic matching techniques to find semantically similar software in other programming languages. This would have the benefit of greatly expanding the search area for similar software and could also perhaps be used for future programming languages.

This thesis focused on using the AML ontology matcher. And although AML came out best in the OAEI contests, it is not tailored towards comparing ontologies extracted from software projects. As AML is very flexible there are multiple options to increase the performance when comparing ontologies from software projects. The fact that software projects in general, and object-oriented software in particular, have a distinct hierarchical structure could easily be exploited by structural matchers to create better matches. This was already explored a bit with the structural matchers in primary mode. Also adding domain-specific knowledge to, for example, the word matcher could probably increase the accuracy of the alignments. And it could be interesting to investigate if adding matchers employing natural language processing techniques could improve the matching process.

Another topic that could be investigated in future research is the configuration for AML. As can be seen in 5.4, the configuration can have an impact on the existing matchers and selectors. It is possible that different projects could benefit from different configuration sets.

References

[1] A. Orso and G. Rothermel, "Software testing: a research travelogue (2000–2014)," in *Proceedings of the on Future of Software Engineering*. ACM, 2014, pp. 117–132, overzicht van testtechnieken en opzetten.

[2] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 85–103.

[3] L. Luo, "Software testing techniques," *Institute for software research international Carnegie mellon university Pittsburgh, PA*, vol. 15232, no. 1-19, p. 19, 2001.

[4] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*. IEEE, 2014, pp. 201–211.

[5] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn, "An orchestrated survey of methodologies for automated software test case generation," *J. Syst. Softw.*, vol. 86, no. 8, pp. 1978–2001, Aug. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2013.02.061

[6] I. Prasetya, "T3i: A tool for generating and querying test suites for java," in *10th Joint Meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2015. [Online]. Available: http://dspace.library.uu.nl/bitstream/handle/1874/321619/950.pdf?sequence=1

[7] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," in *ACM Int. Conf. on Functional Programming (ICFP)*, 2000.

[8] T. Tervoort and I. Prasetya, "Apsl: A light weight testing tool for protocols with complex messages," in *Haifa Verification Conference*. Springer, 2017, pp. 241–244.

[9] J. Tretmans and E. Brinksma, "Torx: Automated model-based testing," in *First European Conference on Model-Driven Software Engineering*, A. Hartman and K. Dussa-Ziegler, Eds., December 2003, pp. 31–43. [Online]. Available: http://doc.utwente.nl/66990/

[10] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software." in *SIGSOFT FSE*, 2011, pp. 416–419.

[11] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queens School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[12] C. Kapser, P. Anderson, M. Godfrey, R. Koschke, M. Rieger, F. Van Rysselberghe, and P. Weißgerber, "Subjectivity in clone judgment: Can we ever agree?" in *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[13] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5. ACM, 2001, pp. 73–88.

[14] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[15] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*. IEEE, 1999, pp. 109–118.

[16] J. Hage, P. Rademaker, and N. van Vugt, "Plagiarism detection for java: a tool comparison," in *Computer Science Education Research Conference*. Open Universiteit, Heerlen, 2011, pp. 33–46.

[17] J.-H. Wang, P. S. Deng, Y.-S. Fan, L.-J. Jaw, and Y.-C. Liu, "Virus detection using data mining techinques," in *Security Technology, 2003. Proceedings. IEEE 37th Annual 2003 International Carnahan Conference on*. IEEE, 2003, pp. 71–76.

[18] Y. Weiss, A. Torralba, and R. Fergus, "Spectral hashing," in *Advances in neural information processing systems*, 2009, pp. 1753–1760.

[19] L. Cai, W. Tong, Z. Liu, and J. Zhang, "Test case reuse based on ontology," in *Dependable Computing, 2009. PRDC'09. 15th IEEE Pacific Rim International Symposium on*. IEEE, 2009, pp. 103–108.

[20] B. Bonilla-Morales, S. Crespo, and C. Clunie, "Reuse of use cases diagrams: An approach based on ontologies and semantic web technologies," *Int. J. Comput. Sci*, vol. 9, no. 1, pp. 24–29, 2012.

[21] S. Dalal, S. Kumar, and N. Baliyan, "An ontology-based approach for test case reuse," in *Intelligent Computing, Communication and Devices*. Springer, 2015, pp. 361–366.

[22] T. Berners-Lee, J. Hendler, O. Lassila *et al.*, "The semantic web," *Scientific american*, vol. 284, no. 5, pp. 28–37, 2001.

[23] J. Heflin *et al.*, "An introduction to the owl web ontology language," *Lehigh University. National Science Foundation (NSF)*, 2007.

[24] E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching," *the VLDB Journal*, vol. 10, no. 4, pp. 334–350, 2001.

[25] S. Castano, A. Ferrara, S. Montanelli, and G. Varese, "Ontology and instance matching," in *Knowledge-driven multimedia information extraction and ontology evolution*. Springer, 2011, pp. 167–195.

[26] P. Shvaiko and J. Euzenat, "Ontology matching: state of the art and future challenges," *IEEE Transactions on knowledge and data engineering*, vol. 25, no. 1, pp. 158–176, 2013.

[27] B. T. Le, R. Dieng-Kuntz, and F. Gandon, "On ontology matching problems," *ICEIS (4)*, pp. 236–243, 2004.

[28] J. Euzenat, P. Shvaiko *et al.*, *Ontology matching*. Springer, 2007, vol. 18.

[29] L. Otero-Cerdeira, F. J. Rodriguez-Martinez, and A. Gmez-Rodriguez, "Ontology matching: A literature review," *Expert Systems with Applications*, vol. 42, no. 2, pp. 949–971, 2015.

[30] A. Von Mayrhauser, R. Mraz, J. Walls, and P. Ocken, "Domain based testing: Increasing test case reuse," in *Computer Design: VLSI in Computers and Processors, 1994. ICCD'94. Proceedings., IEEE International Conference on*. IEEE, 1994, pp. 484–491.

[31] S. Patel and R. K. Kollana, "Test case reuse in enterprise software implementation–an experience report," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 99–102.

[32] R. Li and S. Ma, "The use of ontology in case based reasoning for reusable test case generation," 2015.

[33] I. Smeureanu and B. Iancu, "Source code plagiarism detection method using protégé built ontologies," *Informatica Economica*, vol. 17, no. 3, p. 75, 2013.

[34] A. Si, H. V. Leong, and R. W. Lau, "Check: a document plagiarism detection system," in *Proceedings of the 1997 ACM symposium on Applied computing*. ACM, 1997, pp. 70–77.

[35] M. K. Shenoy, K. Shet, and U. D. Acharya, "Semantic plagiarism detection system using ontology mapping," *Advanced Computing*, vol. 3, no. 3, p. 59, 2012.

[36] F. B. Aydemir and F. Dalpiaz, "Towards aligning multi-concern models via nlp," in *2017 IEEE 25th International Requirements Engineering Conference Workshops (REW)*. IEEE, 2017, pp. 46–50.

[37] A. Kuhn, S. Ducasse, and T. Girba, "Enriching reverse engineering with semantic clustering," in *Reverse Engineering, 12th Working Conference on*. IEEE, 2005, pp. 10–pp.

[38] C. Ieva, A. Gotlieb, S. Kaci, and N. Lazaar, "Discovering program topoi through clustering," in *32nd AAAI Conference on Artificial Intelligence*, 2018.

[39] M. Achichi, M. Cheatham, Z. Dragisic, J. Euzenat, D. Faria, A. Ferrara, G. Flouris, I. Fundulaki, I. Harrow, V. Ivanova, E. Jiménez-Ruiz, E. Kuss, P. Lambrix, H. Leopold, H. Li, C. Meilicke, S. Montanelli, C. Pesquita, T. Saveta, P. Shvaiko, A. Splendiani, H. Stuckenschmidt, K. Todorov, C. Trojahn, and O. Zamazal, "Results of the ontology alignment evaluation initiative 2016," in *OM 2016 : proceedings of the 11th International Workshop on Ontology Matching co-located with the 15th International Semantic Web Conference (ISWC 2016) Kobe, Japan, October 18, 2016*, vol. 1766. Aachen: RWTH, 2016, pp. 73–129. [Online]. Available: http://ub-madoc.bib.uni-mannheim.de/41576/

[40] I. F. Cruz, F. P. Antonelli, and C. Stroe, "Agreementmaker: efficient matching for large real-world schemas and ontologies," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1586–1589, 2009.

[41] D. Faria, C. Pesquita, E. Santos, M. Palmonari, I. Cruz, and F. Couto, *The AgreementMakerLight ontology matching system*, ser. Lecture

Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2013, vol. 8185 LNCS, pp. 527–541.

[42] G. Ganapathy and S. Sagayaraj, "To generate the ontology from java source code owl creation," 2011.

[43] F. M. Couto, M. J. Silva, and P. M. Coutinho, "Finding genomic ontology terms in text using evidence content," *BMC bioinformatics*, vol. 6, no. 1, p. S21, 2005.

[44] Y. Sun, L. Ma, and S. Wang, "A comparative evaluation of string similarity metrics for ontology alignment," *JOURNAL OF INFORMATION &COMPUTATIONAL SCIENCE*, vol. 12, no. 3, pp. 957–964, 2015.

Appendices

A Ontology

```
<?xml version="1.0"?>
<rdf:RDF xmlns="#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:xml="http://www.w3.org/XML/1998/namespace"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
    <owl:Ontology/>
    <owl:ObjectProperty rdf:about="http://www.semanticweb.org/tt/ontologies/method2/#isPartOfClass">
        <rdfs:domain rdf:resource="http://www.semanticweb.org/tt/ontologies/method2/#Method"/>
        <rdfs:range rdf:resource="http://www.semanticweb.org/tt/ontologies/method2/#Class"/>
    </owl:ObjectProperty>
    <owl:ObjectProperty rdf:about="http://www.semanticweb.org/tt/ontologies/method2/#isPartOfMethod">
        <rdfs:domain rdf:resource="http://www.semanticweb.org/tt/ontologies/method2/#Parameter"/>
        <rdfs:range rdf:resource="http://www.semanticweb.org/tt/ontologies/method2/#Method"/>
    </owl:ObjectProperty>
    <owl:ObjectProperty rdf:about="http://www.semanticweb.org/tt/ontologies/method2/#isPartOfPackage">
        <rdfs:domain rdf:resource="http://www.semanticweb.org/tt/ontologies/method2/#Class"/>
        <rdfs:range rdf:resource="http://www.semanticweb.org/tt/ontologies/method2/#Package"/>
    </owl:ObjectProperty>
    <owl:ObjectProperty rdf:about="http://www.semanticweb.org/tt/ontologies/method2/#isPartOfProject">
        <rdfs:domain rdf:resource="http://www.semanticweb.org/tt/ontologies/method2/#Package"/>
        <rdfs:range rdf:resource="http://www.semanticweb.org/tt/ontologies/method2/#Project"/>
    </owl:ObjectProperty>
    <owl:Class rdf:about="http://www.semanticweb.org/tt/ontologies/method2/#Class">
        <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
        <owl:disjointWith rdf:resource="http://www.semanticweb.org/tt/ontologies/method2/#Method"/>
        <owl:disjointWith rdf:resource="http://www.semanticweb.org/tt/ontologies/method2/#Package"/>
        <owl:disjointWith rdf:resource="http://www.semanticweb.org/tt/ontologies/method2/#Parameter"/>
        <owl:disjointWith rdf:resource="http://www.semanticweb.org/tt/ontologies/method2/#Project"/>
    </owl:Class>
    <owl:Class rdf:about="http://www.semanticweb.org/tt/ontologies/method2/#Method">
        <owl:disjointWith rdf:resource="http://www.semanticweb.org/tt/ontologies/method2/#Package"/>
        <owl:disjointWith rdf:resource="http://www.semanticweb.org/tt/ontologies/method2/#Parameter"/>
        <owl:disjointWith rdf:resource="http://www.semanticweb.org/tt/ontologies/method2/#Project"/>
    </owl:Class>
    <owl:Class rdf:about="http://www.semanticweb.org/tt/ontologies/method2/#Package">
        <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
        <owl:disjointWith rdf:resource="http://www.semanticweb.org/tt/ontologies/method2/#Parameter"/>
        <owl:disjointWith rdf:resource="http://www.semanticweb.org/tt/ontologies/method2/#Project"/>
    </owl:Class>
    <owl:Class rdf:about="http://www.semanticweb.org/tt/ontologies/method2/#Parameter">
        <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
        <owl:disjointWith rdf:resource="http://www.semanticweb.org/tt/ontologies/method2/#Project"/>
    </owl:Class>
    <owl:Class rdf:about="http://www.semanticweb.org/tt/ontologies/method2/#Project">
        <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
    </owl:Class>
</rdf:RDF>
```