Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και
Υπολογιστών

Ανάλυση της σημαντικότητας σχέσεων πηγαίου κώδικα για την εξόρυξη της αρχιτεκτονικής συστήματος

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΣΤΑΥΡΟΠΟΥΛΟΥ ΙΩΑΝΝΑ

Επιβλέπων : Κώστας Κοντογιάννης
Αναπληρωτής Καθηγητής Ε.Μ.Π

Αθήνα, Ιούλιος 2015

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και
Υπολογιστών

Ανάλυση της σημαντικότητας σχέσεων πηγαίου κώδικα για την εξόρυξη της αρχιτεκτονικής συστήματος

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΣΤΑΥΡΟΠΟΥΛΟΥ ΙΩΑΝΝΑ

Επιβλέπων : Κώστας Κοντογιάννης
Αναπληρωτής Καθηγητής Ε.Μ.Π

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 2η Ιουλίου 2015.

.....	.....	.....
Κώστας Κοντογιάννης Αν. Καθηγητής Ε.Μ.Π	Ιωάννης Βασιλείου Καθηγητής, Ε.Μ.Π.	Ying Zou Αν. Καθηγήτρια, Πανεπιστήμιο Queen's

Αθήνα, Ιούλιος 2015

Σταυροπούλου Ιωάννα

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Η εξόρυξη αρχιτεκτονικής είναι μια περιοχή της Τεχνολογίας Λογισμικού που έχει προσελκύσει το ενδιαφέρον πολλών ερευνητών και διάφορες προσεγγίσεις έχουν προταθεί. Ωστόσο, εκτεταμένη έρευνα δεν έχει διεξαχθεί για την επίδραση των σχέσεων που εξάγωνται από τον πηγαίο κώδικα στην εξόρυξη αρχιτεκτονικής. Ο στόχος της παρούσας διπλωματικής εργασίας είναι να πειραματιστεί σχετικά με το αντίκτυπο των σχέσεων που εξάγωνται από τον πηγαίο κώδικα στην εξόρυξη αρχιτεκτονικής. Σε αυτή την κατεύθυνση, διερευνομε το πρόβλημα του εντοπισμού και της επιλογής ενός ελάχιστου συνόλου σχέσεων που μπορεί να χρησιμοποιηθεί για την εξόρυξη, μέσω της ομαδοποίησης, μια αρχιτεκτονικής που είναι αρκετά κοντά σε μία αρχιτεκτονική που θεωρείται κοντά στην πραγματική αρχιτεκτονική ενός συστήματος λογισμικού. Στο πλαίσιο αυτής της διπλωματικής εργασίας, έχουμε εφαρμόσει ένα πλαίσιο που εξάγει σχέσεις από τον πηγαίο κώδικα ενός λογισμικού συστήματος, δημιουργεί ένα στιγμιότυπο του μοντέλου μιας αρχιτεκτονική για διαφορετικούς συνδυασμούς των σχέσεων, ορίζει μια απόσταση ομοιότητας μεταξύ των μοντέλων αρχιτεκτονικής και αξιολογεί την απόσταση ομοιότητας μεταξύ της εξάγώμενης αρχιτεκτονικής και μιας αρχιτεκτονικής που θεωρείται κοντά στην πραγματική αρχιτεκτονική του συστήματος. Τα αποτελέσματα μας αξιολογήθηκαν με την πραγματοποίηση πειραμάτων σε μία ευρεία συλλογή συστημάτων, που ταξινομούνται σε διαφορετικά πεδία εφαρμογής και γλωσσών προγραμματισμού. Εργαστήκαμε με τεχνικές για τη μοντελοποίηση αρχιτεκτονικών λογισμικού, τεχνικές για την εξαγωγή συγκεκριμένων μοντέλων αρχιτεκτονικής από πηγαίο κώδικα, καθώς και τεχνικές για να υπολογίστει με μετρικές η ομοιότητα μεταξύ των μοντέλων αρχιτεκτονικής.

Λέξεις κλειδιά

Τεχνολογία Λογισμικού, Αρχιτεκτονική Συστημάτων, Εξόρυξη Αρχιτεκτονικής,

Abstract

Architecture recovery is an area of Software Engineering that has attracted the interest of many researchers and several approaches have been proposed. However, research has not been conducted on the impact that source code extracted relations have on architectural extraction. The objective of this diploma thesis is to experiment on the impact of source code extracted relation on architectural recovery. In this direction, we investigate the problem of identifying and selecting a minimal set of relations that can be used to extract, through clustering, an architecture that is close enough to the real architecture of a software system. In the context of this thesis, we implemented a framework that extracts relations from source code a software system, generates an instance of the an architecture model for different combinations of relations, defines a similarity distance between architecture models and evaluates a similarity distance between the extracted architecture and a gold standard architecture of the system. The results were evaluated by conducting experiments in a wide collection of systems, classified in different application domains and programming language paradigms. We worked with several techniques for modeling software architectures, techniques to extract instances of concrete architecture models from legacy code, as well as techniques to compute similarity metrics between architecture models.

Key words

Software Engineering, Software Architecture, Architectural Extraction, Reverse Engineering

Ευχαριστίες

Θέλω να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή αυτής της διπλωματικής εργασίας κ. Κώστα Κοντογιάννη για τη συνεχή καθοδήγηση του και την εμπιστοσύνη που μου έδειξε καθώς και τις πολύτιμες συμβουλές του. Δεν θα βρισκόμουν στην πορεία που είμαι σήμερα χωρίς την βοήθειά του. Θέλω να ευχαριστήσω ακόμη τους συνεργάτες μου στο Εργαστήριο Τεχνολογίας Λογισμικού που μοιράστηκαν μαζί μου τις πολύτιμες γνώσεις τους. Ευχαριστώ ακόμη τον Γιάγκο Μυτιλήνη για την υποστήριξη και την υπομονή που μου έχει δείξει. Θα ήθελα τέλος να ευχαριστήσω την οικογένειά μου και κυρίως τους γονείς μου, οι οποίοι με υποστήριξαν και έκαναν δυνατή την απερίσπαστη ενασχόλησή μου τόσο με την εκπόνηση της διπλωματικής μου, όσο και συνολικά με τις σπουδές μου.

Σταυροπούλου Ιωάννα,

Αθήνα, 2η Ιουλίου 2015

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-42-14, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Ιούλιος 2015.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Prof. Kostas Kontogiannis for the continuous support of my diploma thesis research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my diploma thesis. Besides my advisor, I thank my fellow lab mates for the stimulating discussions that we had and the expertise that they shared with me in the last two years . Also I would like to thank Giagkos Mytilinis who supported me though this venture. Last but not the least, I would like to thank my family, especially my parents who have supported me through my life and made it possible for me to be here at this moment.

Stavropoulou Ioanna,

Athens, July 2, 2015

This thesis is also available as Technical Report CSD-SW-TR-42-14, National Technical University of Athens, School of Electrical and Computer Engineering, Department of Computer Science, Software Engineering Laboratory, July 2015.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Architecture Recovery

The cycle of development for large-scale software systems usually begins with requirement gathering, followed by architecture construction and high level design. Then the system is implemented and finally, as part of its life cycle, as the system is maintained, it evolves.

As architecture of a software system, we define the structure of the system, which comprises the software elements, externally visible properties of those elements, and relationships among them [Bass13]. Software architecture design is concerned with gross organization and global control structure of a system. Architecture bridges the gap between the requirements and implementation of the system. Software architecture is a very important concern due to understanding, analysis, reusability, evolution and management of legacy systems.

As a software system evolves so does its implementation and its architecture as well. However, in reality software engineers have to deal with the problem of ensuring that a software system's architecture is kept up to date with its implementation.

In software architecture research, this process of mapping a system's implementation to its architecture is known as software architecture recovery. Architecture recovery is really important for large and complex systems. Consistency between the architecture and the implementation is crucial. This problem is solved by recovering the software architecture from the implementation and comparing the recovered architecture with a gold standard. This is a difficult and time consuming process. Therefore several automated tools have been developed to aid software engineers tackle this obstacle. These tools are based on several areas of software engineering, artificial intelligence, programming languages etc. The most used approaches are Cluster based [Lung98], Domain based [DeBa94] and Structural based [Kosc06].

Finally, because this area greatly concerns the research community of software engineers, several techniques have been proposed [Maqb07], [Cora11], [Garc11] for automated architectural recovery. As a result, several studies have been conducted to evaluate all these techniques that have been proposed, [Wu05a], [Garc13].

Specifically on cluster based architectural extraction, although researchers continually try to come up with better clustering algorithms or ways to fine tune the existing ones, extensive research has not been done on investigating the impact that different relations have on architectural extraction. There is no experimentation on how the input relations affect the outcome of the clustering and which relations should be used. This was our motivation for this work.

1.2 Objective

The objective of this diploma thesis is to further experiment on the impact of source code extracted relations on architectural recovery. In this direction, we investigate the problem of identifying and selecting a minimal set of relations that can be used to extract, through clustering, an architecture that is close enough to the real architecture of a software system.

In the context of this thesis, we will implement a framework that extracts relations from the source code of a software system, generates an instance of an architecture model for different combinations of relations, defines a similarity distance between architecture models and evaluates a similarity distance between the extracted architecture and a gold standard architecture of the system. The results will be evaluated by conducting experiments in a wide collection of systems, classified in different application domains and programming language paradigms.

We will work with several techniques for modeling software architectures, techniques to extract instances of concrete architecture models from legacy code, as well as techniques to compute similarity metrics between architecture models.

Once this thesis is completed we want to be able to answer the following questions:

- Does the selection of source code extracted relations that are used for architectural recovery affect the result of the recovery?
- Can we use only a subset of the relations that exist in the source code and still get an accurate result?
- Which combinations of relations produce the closest architecture to the gold standard?
- What is the smallest subset of relations that can be used and still produce an accurate architecture?
- Is there a right balance between the best possible outcome and the less effort expended to extract relations?

1.3 Thesis Organization

The rest of the thesis is organized in six chapters. The second chapter summarizes the existing background knowledge on which we rely in order to design and implement our framework. The third chapter describes briefly our proposed methodology and modeling tools. The forth chapter presents our similarity algorithm that defines a similarity distance between architecture models and in the fifth chapter we go more into detail about the implementation of our framework. Finally, the sixth chapter presents the experimental infrastructure and methodology and our results. The seventh and final chapter presents the conclusions of our work as well as recommendations for future work on the field.

More specifically, in Chapter 2, we will provide background knowledge firstly on source code analysis and the tools that are used. Then, we will refer to software clustering, and two very well known tools for clustering, ACDC and Bunch. Finally, we will discuss about the Meta Object Facility, a standard for model specification.

Having examined briefly the most important concepts that we will deal with, in Chapter 3 we will get into detail about the methodology that we propose and the models that we designed. We will firstly describe in detail the process outline of our framework and then

we will present the relation model and the architecture metamodel that we designed in order to represent the source code extracted relations and the architecture of a software system respectively.

In Chapter 4, we will describe in detail the similarity algorithm that is used to compare architecture instances and extract the similarity distance of them. We will explain how different types of elements are matched and based on what criteria. Furthermore, we will explain how the similarity metric is computed and what is its physical meaning.

In Chapter 5, we will explain how we chose the clustering methodology that we used, we will present the process of the population of the architecture metamodel and we will provide specific examples to make the process more comprehensible. In addition to that, the examples will also demonstrate practically the use of the similarity algorithm that is described in Chapter 4.

In Chapter 6, we will analyze our experimentation results. We will present the experimental infrastructure that was used and give details about the systems that were analyzed. Then we will display our results with extensive interpretation.

Finally, in Chapter 7, we will refer to the conclusions that we reached during our experimentation with the use of source code extracted relations for architectural recovery. We will also underline some issues that we feel require further exploration.

Chapter 2

Background Knowledge

2.1 Source Code Analysis

The last decades have seen an explosive growth in the software industry worldwide. This development has put in the spotlight the need of tools that analyze large software systems. Developers need tools to discover and map unknown source code for enhancement or maintenance tasks. Tools that are helpful in this direction:

- give information about metrics of the source code, that provide quantitative indicators to identify maintenance hot spots
- offer visualization that can help the understanding of the system composition
- detect patterns
- perform dependency analysis that reveal component interactions

When all these tools are combined, the developer has a complete view of the system and its interacting components.

Several fact extraction tools exist that offer most or all of these features. [Bois07] compares several C++ fact extractors and we will present their findings. Table 2.1 compares some fact extractors based on the features that we previously described.

The abbreviations that are used are: (i) MC for Metric Calculation; (ii) PD for Pattern Detection; (iii) DA for Dependency Analysis; and V for Visualization. It should be noted that the fact that a fact extractor lacks one or more of these features only means that it was designed differently and has other key features. It is also indicated whether the extractor is open source, which is also an important aspect in the context of this thesis.

Open Visualization Toolkit - Combines Source Navigator [SN] with Open Inventor C++ toolkit [Tele02]. This tool has been mainly used for reverse architecting large systems.

TkSee/SN -Also uses Source Navigator [SN] to generate a GXL representation of the Dagstuhl Middle Model (DMM) [McQu06]. This tool supports navigation as well as querying. However, even though this tool is indicated in [Leth02] to be open source, no reference

Fact Extractor	MC	PD	DA	V	Open Source
Open Visualization Toolkit	√			√	
TkSee/SN	√		√		√
SN/Rigi	√		√	√	Partially
CPPX	√	√	√	√	√
Fetch	√	√	√	√	√

Table 2.1: Comparison of Fact Extractors

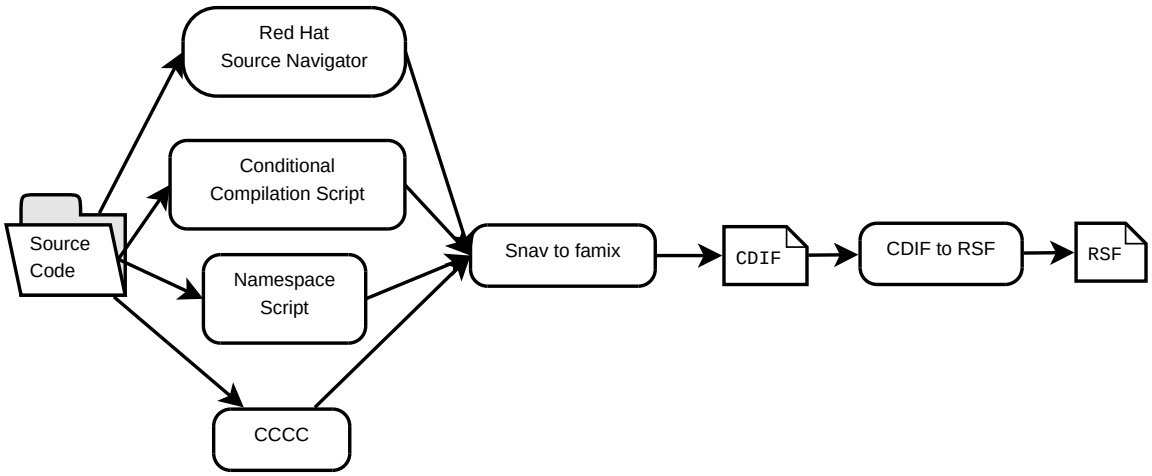

Figure 2.1: Fact Extraction Tool Chain

to the source code nor any binary release exists.

SN/Rigi - In [Mois03] a C++ fact extractor is presented that combines Source Navigator [SN] with Rigi [Mull88]. The output model comforms to a C/C++ domain model and is presented in a Rigi Standard Format (RSF).

CPPX - CPPX extracts facts from C++ source from the highest semantic level (classes and global data and functions) down to the lowest code level of individual statements and expressions. The output format is a GXL (Graph eXchange Language) representation of Datrix [Dean01].

Fetch - Fetch (Fact Extraction Tool CHain) [Bois07] is a tool chain for software analysis. It consists of several open-source tools targeting the exploration of large C, C++ and Java software systems. It is open source and offers several tools to work with.

From the extraction tools we previously described, we chose to work with Fetch because it offers good documentation which made it easier to extend and also can extract facts from C, C++ and Java systems, so we can have a variety in the systems that we examine for our experiments.

Figure 2.1 demonstrates how Fetch combines different tools and extracts the desired output. Details about each tool will be discussed next.

Fetch takes as input the source code of the system under examination. Source code does not have to be compilable or preprocessed in any way. The first tool that is applied on the source code is Source Navigator [SN]. Source Navigator was initially developed by Red Hat and is a source code analysis and comprehension tool that provides an IDE for understanding and reengineering large software systems. Source Navigator performs fast lexical analysis, extracting information from C, C++, Java and other systems and ten uses this information to build a system database. Source Navigator graphical browsing tools use this database to query symbols, such as functions and global variables, and the relations between them. Source Navigator does not provide low-level and detailed information about the system under examination which makes it fast and ideal for architectural analyses.

In addition to Source Navigator, another tool, C and C++ Code Counter, CCCC [Sc01], is used to extract information that Source Navigator is unable to deduct from the structure of the source code, such as Lines of Code, Lines of Comments and Cyclomatic Complexity. Finally two home-made scripts run on the source code, one to extract namespace scopes and the other to extract conditional compilation directives.

Entity	Origin
File, Include	Source Navigator
Conditional Compilation	Conditional Compilation Script
Package	Namespace Script
Class, Inheritance, Typedef	Source Navigator
Method, Function	Source Navigator
Attribute, Global Variable	Source Navigator
Invocation, Access	Source Navigator and Snavtofamix
Measurement	Source Navigator and CCCC

Table 2.2: Abstract model content

The facts extracted by these tools are combined and interconnected by Snavtofamix, which queries the created database and resolves cross-references such as inheritance, invocations, accesses etc. The output is an abstract model corresponding to the Famix (FAMOOS [famo] Information Exchange Model) specification. The representation of the Famix model is in a Case Data Interchange Format, CDIF [Imbe91]. Table 2.2 presents the contents of the Famix model as well as the tool that extracts each one from the source code.

The final step is to translate the abstract model that was created to a graph format (RSF, Rigi Standard Format). This transformation is implemented by the CDIF2RSF tool and is required to facilitate querying the abstract model and extracting specific information.

Fetch offers several other tools for visualization and graph querying, such as Guess, R scripts and Crocopat, but those tools will not be further discussed since they are out of the scope of this thesis and were not used.

To make everything that was discussed more clear, a simple example of the use of Fetch is presented next.

In Listing 2.1, a small C program that prints a Pascal Triangle is presented.

```
#include <stdio.h>
long factorial(int);
int main(){
   int i, n, c;

   printf("Enter the number of rows to see in pascal triangle\n");
   scanf("%d",&n);

   for (i = 0; i < n; i++) {
      for (c = 0 ; c <= i; c++)
         printf("%ld ",factorial(i)/(factorial(c)*factorial(i-c)));
      printf("\n"); }
   return 0;
}

long factorial(int n){
   int c;
   long result = 1;

   for (c = 1; c <= n; c++)
         result = result*c;
   return result;
}
```

Listing 2.1: Pascal Triangle

This source code is the input of Fetch. Source Navigator, CCCC and the home-made scripts run on this input, the database is created, however we do not have access to it. Snavtofamix is also executed, resulting to the creation of CDIF, which is presented in Listing 2.2.

Lines 1–10 represent the header of the model and contain general information about the program such as the language of the source code and the name of the system (which comes from the name of the folder that was given as input to Fetch) and when Fetch was executed. These lines are followed by the abstract model.

Lines 12–16 represent a source file. The field *uniqueName* provides the unique name of the file, which is different from the *name* field only when there are several same-named files in the project, which is not the case. If there were more source files in the system, these lines would be repeated for each one of them.

In addition to that, information about the library that is included is given in lines 18–22. The information that is includes is: which file (line 19) includes which other file (line 20) and at what line of the source code this occurs (line 21).

Lines 24–30 and 39–45 are about the two functions of the source code, main() and factorial(), and provide information such as the names and signatures of the functions and their return types or classes.

Also, because the factorial() function is defined in the file pascal.c, CDIF contains lines 32–37, which state where the function is declared.

Finally, the main() function calls factorial() and therefore there is an invocation in lines 47–53. Line 48 defines the function that causes the invocation and line 49 defines the function that is invoked. In addition to that, there is the line where the invocation occurs (line 50), as well as where the above mentioned functions are declared(lines 51–52).

```
(:HEADER
  (:SUMMARY
    (ExporterName "snavtofamix")
    (ExporterVersion "1096")
    (ExporterDate "2015/05/27")
    (ExporterTime "14:44:24")
    (ParsedSystemName "pascalFolder")
    (SourceLanguage "C"))
)

(:MODEL
(SourceFile FM1
  (uniqueName "pascal.c")
  (name "pascal.c")
)

(Include FM2
  (includingFile "pascal.c")
  (includedFile "stdio.h")
  (sourceAnchor #[file "pascal.c" start 1 end 1|]#)
)

(Function FM3
  (name "factorial")
  (signature "factorial(int)")
  (declaredReturnType "long")
  (declaredReturnClass "")
  (sourceAnchor #[file "pascal.c" start 3 end 3|]#)
)

(FunctionDefinition FM4
  (name "factorial")
  (declaredBy "factorial(int)")
  (sourceAnchor #[file "pascal.c" start 26 end 26|]#)
  (declSourceAnchor #[file "pascal.c" start 3 end 3|]#)
)

(Function FM5
  (name "main")
  (signature "main()")
  (declaredReturnType "int")
  (declaredReturnClass "")
  (sourceAnchor #[file "pascal.c" start 5 end 5|]#)
)

(Invocation FM6
  (invokedBy "main()")
  (invokes "factorial(int)")
  (sourceAnchor #[file "pascal.c" start 18 end 18|]#)
  (sourceSourceAnchor #[file "pascal.c" start 5 end 5|]#)
  (destinationSourceAnchor #[file "pascal.c" start 3 end 3|]#)
)
)
```

Listing 2.2: CDIF for Pascal Triangle

Once CDIF is created it can be transformed to RSF in order to be more useful. By running the CDIF2RSF script an RSF file is created. This file is presented in Listing 2.3. We should note that CDIF contains information that is not included in the RSF file, for example the fact that stdlib.h is included in pascal.c. This happens because standard system libraries that are included are not presented in the RSF. If pascal.c included a non standard system library, the RSF file would have had another line to describe this fact.

```
FileBelongsToModule "pascal.c"#1 "/"#M1
DeclaredIn "factorial(int)"#3 "pascal.c"#1
DefinedIn "main()"#5 "pascal.c"#1
DeclaredIn "main()"#5 "pascal.c"#1
DefinedIn "factorial(int)"#3 "pascal.c"#1
Calls "main()"#5 "factorial(int)"#3
```

Listing 2.3: RSF for Pascal Triangle

If we take a closer look at the RSF file, it contains triples in the form: *Relation* Entity Entity. Each line represents a different relation, there are no duplicates and the Entities are characterized by their name, followed by a unique identifier.

This was a very simple example, with one source file that has two functions and just one call. Real software systems contain hundreds of source files and much more complex associations between them. However, the procedure that was described is the same for systems of any kind and any size.

Finally, we will take a look at the relations that Fetch is able to extract from the source code. These are presented on Table 2.3 with a small description for each one of them. There are several types of relations and apparently for our purpose not all these relations are useful. We will elaborate on the relations that we use on Section 3.2.

	Relation	**Description**
Containment	Module Belongs to Module	States a containment relation between two modules
	File Belongs to Module	States to which Module each File belongs
	Class Belongs to File	States to which File each Class or Struct belongs
	Invocable Entity Belongs to File	States the Global Variables that belong to each File
	Method Belongs to Class	States the Methods that belong to each Class
	Attribute Belongs to Method	States the Attributes of each Method or Struct
Macros	Macro Definition	States the definition of a Macro as well as the file where it is defined
	Macro Use	States in which File a Definition, that was previously defined, is used
Conditions	Conditional Compilation	States under which conditions a block of code is compiled
Location	Entity Location	States in which File and from which to which line an Entity is located
	Entity Belongs to Block	States to which block of source code – which is conditionally compiled – an Entity belongs

Declaration	Defined In	States in which file a function or a method is defined. This is the file where the actual body of the function or the method is located. We should underline that for every function and method there is only one *Defined In* relation in the output file of the extractor.
	Declared In	This relation is similar to the one above, with one great difference. *Declared In* states in which file there is a declaration of the function or the method. As a result there can be more than one such relations for the same function or method in the output file of the extractor.
Invocations	Call	Declares an invocation between two methods or functions
Types	Type Definition	This relation states that a type name is associated with a class, a struct or another type name that was previously defined
	Uses Type	This relation states which class or struct, not a primitive type, is used in a method or a function respectively
	Has Type	States the type of a function, method, attribute or global variable.
	Has Type Definition	This relation combines the previous ones and associates a function, method, attribute or global variable with a type name
Information	Signature	States the signature of a method
	Visibility	States the visibility of a class, attribute or method when this is applicable
	No of Lines	States in which line of a File an Entity belong to
Other	Inherits From	Represents the inheritance of classes in object-oriented programming
	Include	States which File is included in another File
	Access	States in which method or function, an attribute or a global variable is on the right side of an assignment statement
	Set Variable	States in which function or method an attribute or a global variable is on the left side of an assignment statement

Table 2.3: Relations that Fetch extracts

2.2 Software Clustering

Software clustering methodologies group entities of a software system into meaningful subsystems in order to help with the process of understanding the high level structure of a large and complex software system. A software clustering approach that is successful in accomplishing this task can have significant practical value for software engineers, particularly those working on legacy systems with obsolete or even non-existent documentation.

Software clustering has troubled the research community for more than two decades. During this time, several software clustering algorithms have been published in the literature, most of which have been applied to particular software systems with success. The key to determining if a clustering algorithm is considered successful, is the distance of the clustering solution that it offers, to that of an expert who is knowledgeable of the system. In this direction the research community has developed several methods to assess the quality of software clustering algorithms such as the MoJo distance [Tzer99], the Craft framework [Mitc01] and several other experiments that have been conducted [Anqu99] and [Kosc00].

Software Clustering has preoccupied the research community for so long, because it solves important software engineering problems in mainly two areas, software evolution and information recovery.

Software Evolution: Every software system evolves in order to add new functionality, correct existing faults and improve maintainability. Software clustering tools attempt to improve the software structure – e.i. software restructuring – or reduce the complexity of large modules – e.i. source code decoupling.

Software restructuring is a form of perfective maintenance that modifies the structure of a program's source code. The goal is to facilitate maintenance activities, such as adding new functionality or correcting previously undetected errors within a software system [Lung04]. Source code decoupling attempts to reduce the complexity of complex modules or functions. In [Lung06] a case study is presented where software clustering is applied for source code decoupling at the procedural level. Software clustering groups related statements together in order to produce dependency rank between the groups.

Information Recovery: The primary goal of reverse engineering is to recover components or to extract system abstractions. Several approaches and techniques have been proposed in the literature. Especially, architecture recovery methods utilize software clustering with success. The architecture recovery methods focus on discovering the system architecture by analyzing abstractions extracted from the source code, such as components, subsystems and design patterns. For example, in [Baue04] design patterns are used to recover architecture or in [Zhao10] software clustering is used to generate the software architecture of business applications.

Software Clustering is also used to identify duplicate code [Lung06] as well as predict the fault proneness of software modules [Zhon04].

It is obvious that software clustering is an important technique, that can be used to solve several problems in our area. We will take a closer look at two software clustering tools, the ACDC (Algorithm for Comprehension Driven Clustering) [Tzer00] and the Bunch clustering suite [Manc99] which are commonly used in experiments in the literature.

2.2.1 ACDC

ACDC is an algorithm that is based on the detection of commonly observed patterns in the subsystems of large software systems. As [Tzer00] underlines, most clustering algorithms aim to achieve low coupling and high cohesion, interface minimization, shared neighbours etc. However, in the effort to maximize performance or accuracy of the algorithms, researchers have forgotten about the primary concern of software clustering, which is comprehension. In this direction, the algorithm uses pattern recognition to find certain system decompositions that appear frequently in software systems. In addition to that, the algorithm uses meaningful cluster names that give information about each cluster and make the decomposition more understandable. Finally, the size of the clusters is bounded in order to make the decomposition even more manageable and comprehensible. The maximum size of each cluster is set by default at 20 elements.

ACDC offers several features to help the software engineer understand the obtained decomposition faster. These features are:

1. *Effective cluster naming*: This allows engineers associated with the software system to understand the decomposition more easily and start benefiting from it faster. However, this problem has not attracted much attention, the researchers that created ACDC believe that it is important for the high level view of a software system to have names associated with the software system, rather than SS01 or Subsystem01.

2. *Bounded cluster cardinality*: Decompositions containing clusters of one or two resources even though they achieve low coupling, are practically unusable. On the other hand, with very large clusters the same problem is presented. Therefore, ACDC bounds cluster sizes to a limited number of objects to make the decomposition more comprehensible and useful. However, this should not be done at the expense of the system's structure and if this is the case, large clusters should be simply further decomposed producing nested clusters of several levels.

3. *Pattern-driven approach*: ACDC is based on patterns that commonly emerge in manual decompositions created by experts. These are patterns that are found in large systems, containing more than 100 source files, covering a wide range of familiar subsystem structures that appear in manual decompositions of large industrial software systems. The patterns that ACDC detects are presented on Table 2.4.

Therefore, ACDC utilizes this knowledge and identifies these patterns in order to create a system decomposition.

The algorithm is executed in two steps. Step one, based on the identification of the predescribed patterns, an initial decomposition is created and the created clusters are named appropriately. Firstly, source file clusters are created and from this point after the file and the contained entities are considered as an atomic entity. Then, the body-header pattern, the leaf collection as well as the support library patterns are identified. For the later, there are no clusters created, simply entities are identified as candidates. Then, the central dispatcher and subgraph domination clusters are created and finally a supporting cluster is created including the remaining elements that were identified as candidates for the support library pattern and have not been assigned to any other subsystem. During step two, the remaining elements that have not been assigned to any subsystem are incorporated into the existing decomposition using the Orphan Adoption technique [Tzer97]. In other words, the remaining elements of the systems that were not put into clusters during the first step are added to the cluster that

Pattern	Description
Source file pattern	The set of procedures/functions/methods can be grouped together with the variables contained in the same source file.
Directory structure pattern	The structure of the source code may correspond to a subsystem decomposition. However, this pattern is not always the case and must be used with caution. Many systems contain folders with a collection of header files or libraries and such a cluster will not be useful from a comprehension point of view.
Body-header pattern	The header file and the corresponding body file, such as .c and .h files in C, can create a cluster containing only these files. Even though this is a really small cluster, it can reduce the complexity of a system's structure.
Leaf collection pattern	A set of files which are not connected to each other, but offer similar services –such as a set of drivers– can be grouped together.
Support library pattern	Procedures accessed by the majority of the system's subsystems [Mül93] are grouped together into one cluster.
Central dispatcher pattern	This pattern is the dual of the support library pattern. Large systems commonly contain a small number of resourses that depend on a large number of other resources. These are ignored initially, in order not to obscure other patterns and then are reconsidered once subsystems have already been formed.
Subgraph dominator pattern	This pattern searches for a subgraph in the system's graph which contains a set of nodes where in order to access each node of the set, one must go through one specific node, considered a "dominator node".

Table 2.4: Patterns detected by ACDC

have the most interaction with. If there are several clusters that comply with this criterion, a new cluster is created and all the elements that can not be put into an existing cluster with certainty are placed there.

2.2.2 Bunch

Bunch clustering suite [Manc99] provides a variety of algorithms for Software Clustering. Bunch takes as input a module dependency graph, e.g.an RSF file, and aims to create a decomposition of the system which meets two criteria:

- Highly interdependent elements are grouped in the same clusters.
- Independent elements are assigned to separate clusters..

One way to solve this problem is to use exhaustive search and find the best of all the possible decompositions for the system. In order to achieve that, a function that determines the quality of a decomposition is required. The function that Bunch uses is the Modularization Quality measurement (Section 2.2.2). Given that function, even though exhaustive search always gives the best answer, the number of decompositions that need to be checked grow very rapidly and a 15-node graph is the limit for performing exhaustive search. Therefore, there is a need for a sub-optimal strategy to solve this problem.

Hill Climbing

Hill Climbing is an other solution. The idea is to find a starting position and with every move transition to a better state. Bunch offers two Hill-Climbing algorithms based on how the choice of the next state is made. Firstly, there is the Steepest Ascent Algorithm which is a greedy algorithm. Steepest Ascent calculates all the possible states it can transition to and takes the best one. As all greedy algorithms, even though each step is optimal, the algorithm may not be optimal overall. Secondly, there is the Nearest Ascent Algorithm, which is a non-greedy Hill Climbing algorithm. Nearest Ascent does not compute the set of all the possible states to transition to, but computes states until a better one than the current is found. As a result, it is much faster than Steepest Ascent. However, there are some common problems with Hill Climbing, such as what happens if we find a local maximum and how to avoid getting a bad start, which also apply to our problem. In order to help Hill Climbing algorithms, Bunch restarts the algorithm at some random state after finding a local maximum. Another technique to assist Hill Climbing algorithms is Simulated Annealing. In this case, the algorithm occasionally takes a transition that is not an increase in quality in the hope that this will lead to climbing a better hill.

Genetic Algorithms

Another solution is Genetic Algorithms. Generally, Genetic Algorithms are heuristics that mimic the process of natural selection, and generate solutions to optimization problems using techniques inspired by natural evolution, such as inheritance, mutation, selection and crossover. Genetic Algorithms work by representing each state as a string and then applying genetic manipulation in ways analogous to nature as we described before. Genetic Algorithms tend to provide good result and converge to a solution quickly when the solutions space is small relative to the search space. Additionally, this approach tends to overcome local maxima that Hill Climbing struggles with. .

Modularization Quality

The Modularization Quality (MQ) is a measurement of the quality of a partition. In each partition there are two types of edges, Intra-Edges which are edges inside one cluster and Inter-Edges which are edges connecting different clusters. For the rest of this section we will refer to Intra-Edges as μ edges and Inter-Edges as ε edges.

Given k clusters in one partition, MQ is calculated by Equation 2.1, where CF_i is the cluster factor for cluster i and is calculated by Equation 2.2, which increases as the cluster's cohesiveness increases.

$$MQ = \sum_{i=1}^{k} CF_i \tag{2.1}$$

$$CF_i = \begin{cases} 0 & \mu_i = 0 \\ \frac{2\mu_i}{2\mu_i + \sum_{j=1 j \neq i}^{k} (\varepsilon_{i,j} + \varepsilon_{j,i})} & \text{otherwise} \end{cases} \tag{2.2}$$

2.3 Architecture Representation

According to the Object Management Group (OMG), a model of a system is a description or a representation of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text, which may either in a modeling or a natural language [OMG]. Since models are used extensively, not only in model-driven and reverse engineering but across most of the areas of Software Engineering, it is imperative that models are represented in a uniform and expressive manner. For this purpose various modeling languages have been proposed. These languages can either be graphical or textual. Graphical modeling languages use a diagram technique with named symbols that represent concepts, lines that connect the symbols and represent relations and various other graphical notations to represent constraints. On the other hand, textual modeling languages use standardized keywords accompanied by parameters or even natural language terms and phrases to make computer-interpretable expressions.

The most well-known modeling language is the Unified Modeling Language (UML) which has established itself as an industry standard for specification, design, visualization as well as documentation of software systems. In UML terms, a model is an instance of the UML metamodel and a diagram describes a graphical representation of a model [Selo03]. Additionally, a model element refers to a UML metaclass instance, while a property of a model element refers to a meta-attribute instance belonging to the model element. The state of a model element is defined by the property values.

2.3.1 The Meta Object Facility (MOF) Standard

The Model Driven Architecture (MDA) is a software design approach for development of software systems. It provides a set of guidelines for structuring of specifications. It was launched by the OMG in 2001 and is related to multiple standards including the Unified Modeling Language (UML), the Meta-Object Facility (MOF), the XML Metadata Interchange (XMI) etc.

As [OMG] mentions, the Meta Object Facility Specification defines an abstract language and a framework for specifying, constructing and managing technology neutral metamodels. A metamodel is in effect an abstract language for some kind of metadata. Examples include

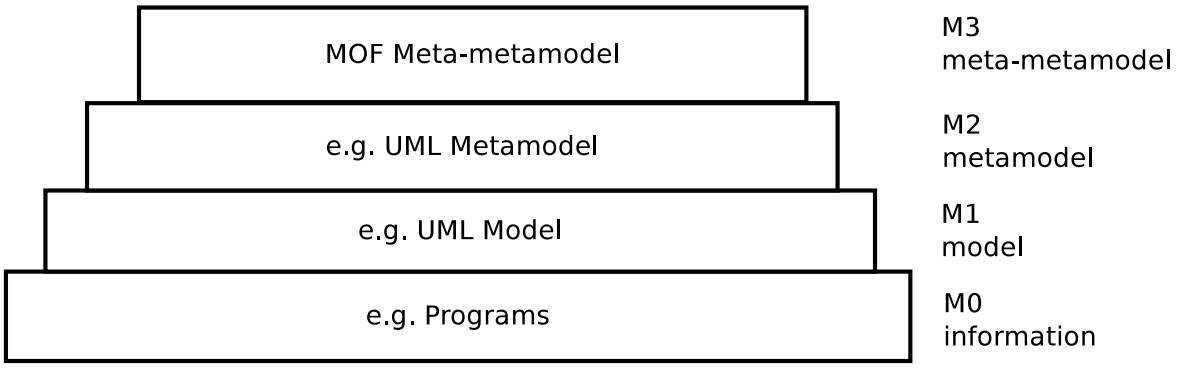

Figure 2.2: Four Layer Metadata Architecture

the metamodels for UML and the MOF itself, as well as those in various OMG specifications in progress.

In addition, MOF defines a framework for implementing repositories that hold metadata (e.g. models) described by the metamodels. This framework uses standard technology mapping to transform MOF metamodels into metadata APIs. This gives consistent and interoperable metadata repository APIs for different vendor products and different implementation technologies.

Four Layer Metadata Architectures

The classical framework for metamodeling is based on an architecture with four meta-layers, where each layer is an instance of the layer above. Figure 2.2 illustrates the classical four layer metamodeling framework.

At the top of the hierarchy of the four layers is the *M3* layer. The primary responsibility of this layer is to define the language for specifying a metamodel. MOF is an example of a meta-metamodel. According to [Objeb], a meta-metamodel is typically more compact than a metamodel that it describes, and often describes several metamodels. It is generally desirable that related metamodels and meta-metamodels share common design philosophies and constructs. However, each layer can be viewed independently of the other layers, and needs to maintain its own design integrity. Meta-metamodels are defined using meta-metamodels, i.e. meta-metamodels are reflective, MOF is defined using MOF itself.

The metamodel layer is an instance of the meta-metamodel layer. The primary responsibility of the metamodel layer, often referred as *M2*, is to define a language for specifying models. UML and the OMG Common Warehouse Metamodel (CWM) are examples of metamodels. Metamodels are typically more elaborate than the meta-metamodels that describe them, especially when they define dynamic semantics. The UML metamodel is an instance of the MOF.

Models belong to the model layer, referred as *M1*. A model is an instance of a metamodel. The primary responsibility of the model layer is to define languages that describe semantic domains. A user model is an instance of the UML metamodel. In the domain of software, the elements of the *M1* layer are the models of the software systems which are defined in UML models (Class diagrams, sequence diagrams etc.).

The hierarchy bottoms out at *M0*, which contains the runtime instances of model elements defined in a model. The snapshots that are modeled at *M1* are constrained versions of the *M0* runtime instances.

It should be underlined that there are architectures with more than four levels, which all

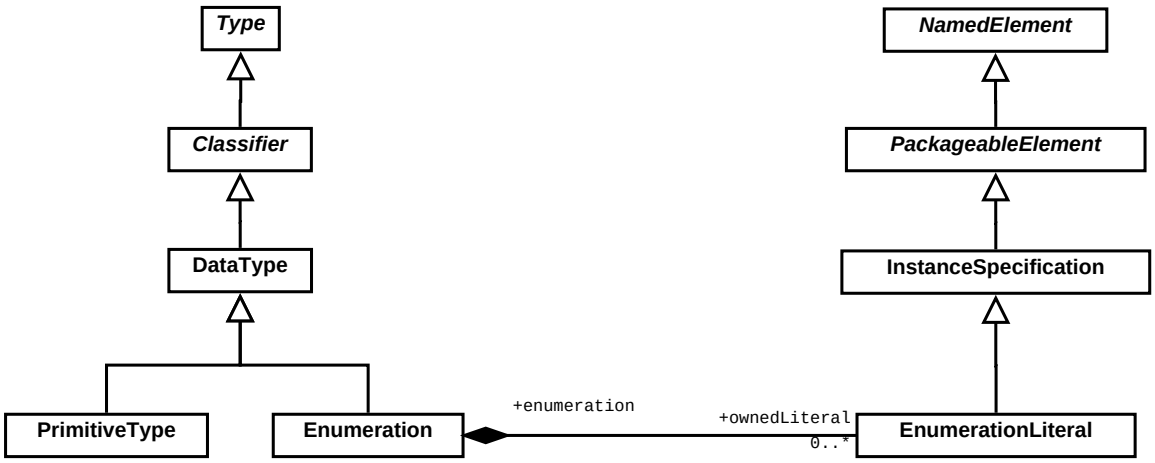

Figure 2.3: EMOF Data Types

share the same characteristic; the existence of pairs of classes-instances and a mechanism to traverse from the instance to its class.

Structure and Description of MOF

The most recent version of MOF (2.4.2) is designed and specified in a way that is tightly connected with UML 2.0. Additionally, for its specification MOF utilizes the syntax and semantics of UML. Actually, for its specification MOF utilizes a subset of UML (mainly class diagrams), an object constraint language (OCL) and precise natural language.

The specification provides two variations of MOF: the Essential MOF (EMOF) and the Complete MOF (CMOF). Both EMOF and CMOF are described using CMOF, which is also used to describe UML2. EMOF is also completely described in EMOF by applying package import, and merge semantics from its CMOF description. As a result, EMOF and CMOF are described using themselves, and each is derived from, or reuses part of, the UML 2.0 Infrastructure Library.

While the purpose of CMOF is to provide a general framework for metamodeling, EMOF is the subset of MOF that closely corresponds to the facilities found in Object Oriented Programming Languages and XML.

As the functionality of EMOF is enough for this thesis, Figures 2.3 - 2.5 present the MOF diagrams that specify EMOF as they are presented in the specification document [Obje13].

It should be underlined that the complete specification of EMOF contains several constraints as well as guidelines and definitions which can be found in the specification document.

XML Representation

MOF describes the means to create and manipulate models and metamodels. However, MOF is a standard and not an implementation. The creation of implementations that correspond to the MOF specification is at the discretion of software producers. MOF may even be particularly useful in a programming environment where different technologies coexist. Therefore, the issue of interconnectivity with other standards is is imperative. This issue is addressed by the specification of mappings to other standard frameworks, such as mapping from MOF to Interface Definition Lanuage (IDL) [Objea], to Java using the Java Metadata Interface (JMI) [Proc02], to XML using the XML Metadata Interchange (XMI) [Obje07] etc.

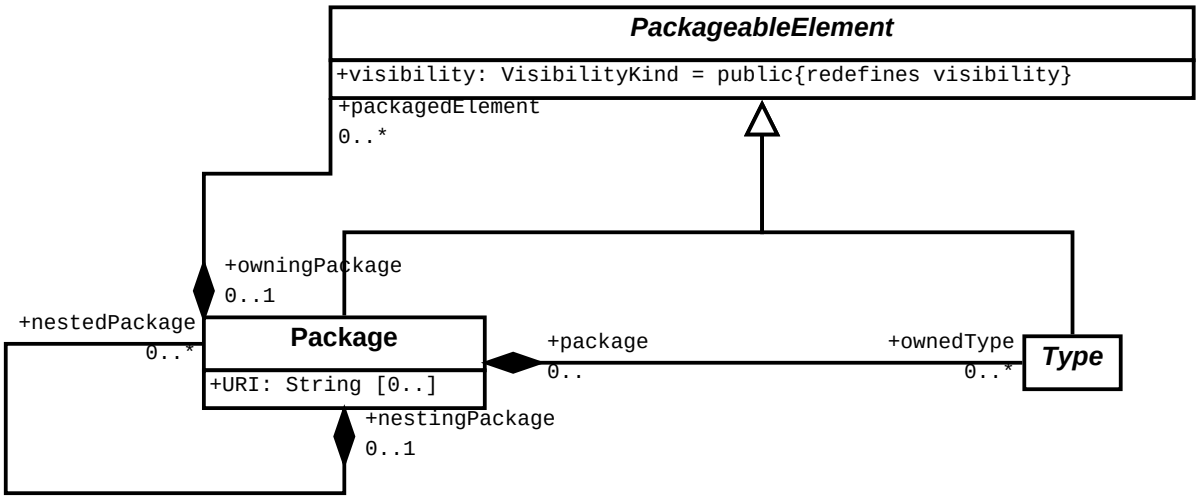

Figure 2.4: EMOF Package

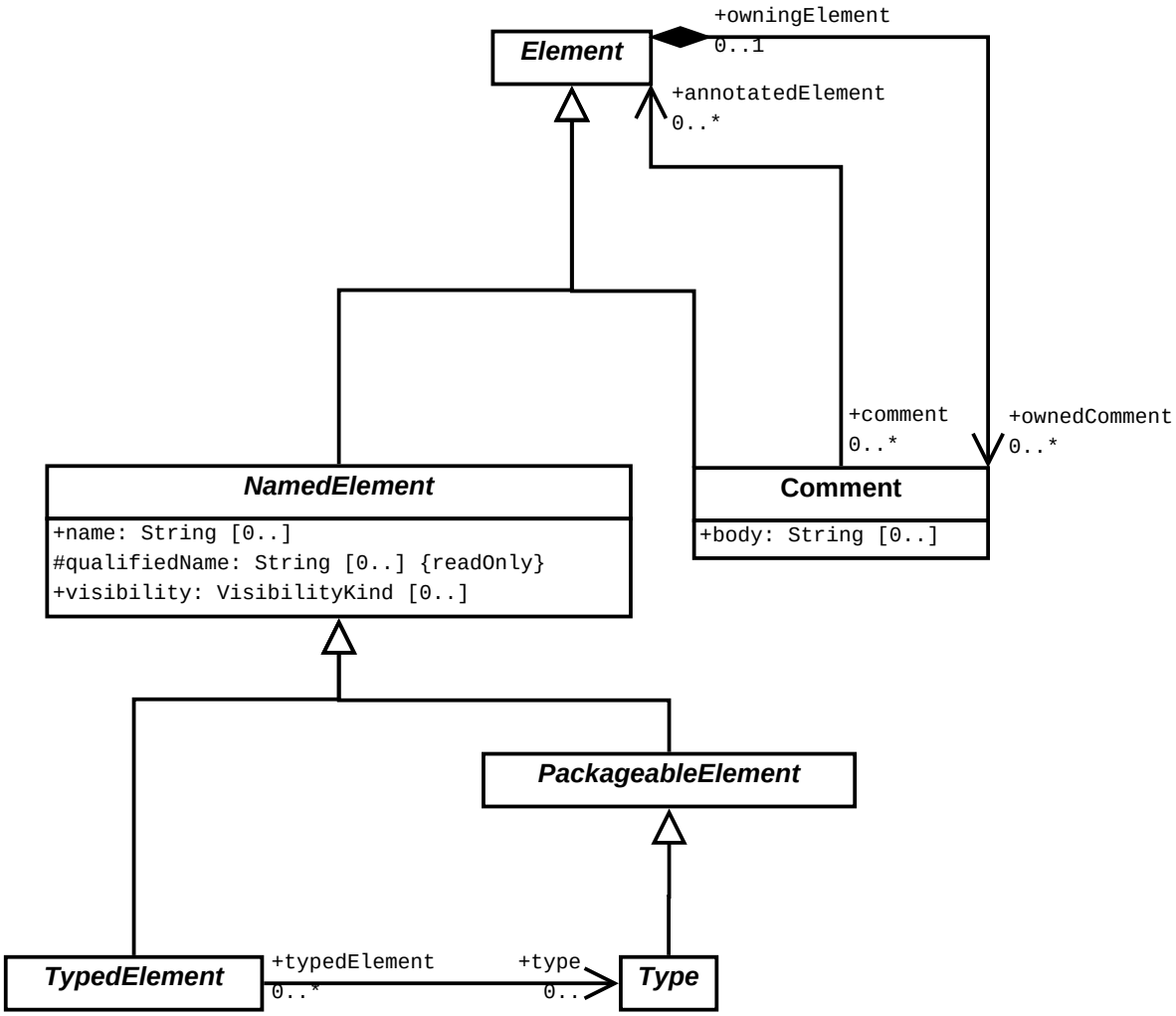

Figure 2.5: EMOF Types

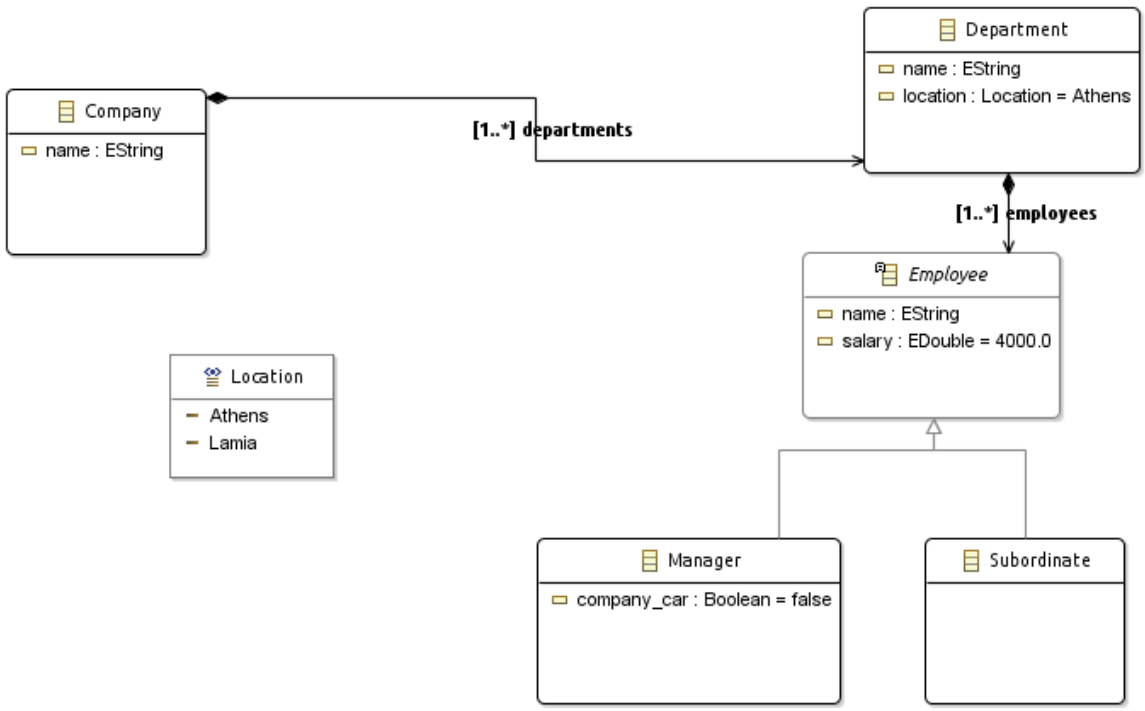

Figure 2.6: Example model (M1)

XMI is an OMG standard for exchanging metadata information via Extensible Markup Language (XML). It can be used for any metadata whose metamodel can be expressed MOF. The most common use of XMI is an interchange format for UML models, although it can also be used for serialization of models of other languages.

Therefore, we have an infrastructure to represent architecture models either with graphical (UML, MOF) or textual (XMI) ways.

For example, Figure 2.6 presents an simple model (M1) for Companies and Listing 2.4 presents a part (the class *Employee*) of the corresponding XMI file it visualizes. This model is an instance of the Ecore (M2) model, which is an implementation of the EMOF(M3).

```
<eClassifiers xsi:type="ecore:EClass" name="Employee" abstract="true">

  <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="
 ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>

  <eStructuralFeatures xsi:type="ecore:EAttribute" name="salary" eType
 ="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EDouble"
      defaultValueLiteral="4000.0"/>

</eClassifiers>
```

Listing 2.4: Example XMI (M1)

Chapter 3

Proposed Methodology and Modeling

3.1 Process Outline

Figure 3.1 illustrates how several tools are combined in the framework that we implemented for our experiments in order to evaluate several source code extracted relations.

The input of the process is the source code of a specific software system, either procedural or object oriented. The source code does not need to be preprocessed in any way.

The first phase consists of analyzing the source code and extracting an RSF file containing all the relations of the relation model discussed in Section 3.2. For this phase we use Fetch that was discussed in Section 2.1 and a script to eliminate the relations that do not belong to the relation model. Once the RSF file is created, using a simple script all the combinations of the relations that belong to the RSF file are generated. The combinations are created in a specific order. First there are the combinations containing one relation, then those containing two relations, then those containing three relations etc. A simple way to understand the generation process of the combinations is to compare it to a counter, in the tridecimal numerical system for example, if the system contain 13 relations. As a result, the number of combinations that are created depends on the number of relations that the system contains. For example, if the system contains 13 relations, we end up with 8192 different combinations of relations, for 12 relations, 4096 combinations etc.

During the next phase, the entities of each of these combinations are clustered using a clustering algorithm. Once the clustering result for a specific combination is available, using this information and the corresponding RSF file we can populate an instance of the Architecture Metamodel (Section 3.3). In a similar way, by taking into consideration all the relations of the system, the architecture instance that is considered the gold standard is created.

In the final phase, all the architecture instances are compared to the gold standard using our similarity algorithm that calculates the distance of the two instances. The output of this final phase is a difference score between the gold standard and the architecture under examination. Based on this score we can evaluate the combination and decide whether it is considered a good, an average or a bad combination. This means that the architecture created by a good combination is close to the real architecture of the system and therefore this combination of relations is useful for architectural recovery. On the other hand, bad combinations do not have enough amount of information and therefore should not be used for architectural extraction.

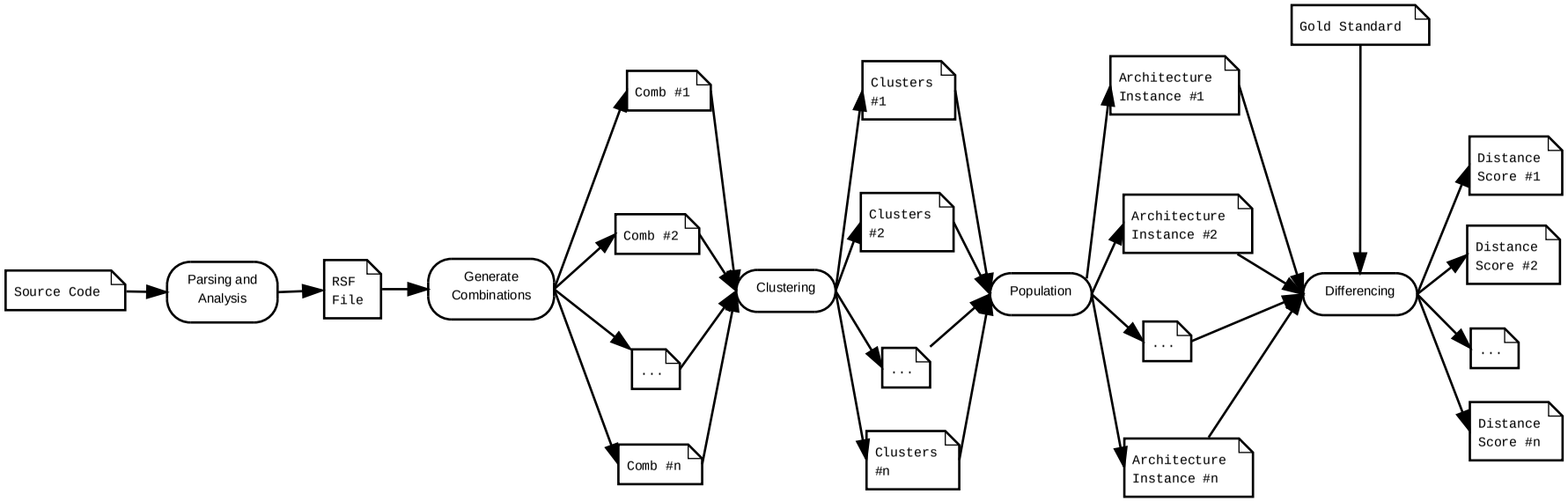

Figure 3.1: The chain of tools comprising the implemented framework

3.2 Relation Model

Once the extraction tool has completed all the steps, the resulted file contains the relations from the source code. The extraction tool can extract 22 different relations from the source code. However, from these relations we only use 13 of them.

The choice of these relations was made based on two criteria. Firstly, the relations that would not give information about the architecture of the system, needed to be eliminated. For example, the information if a method is abstract or what is its signature would not add information that the clustering could really utilize and would not make the extracted architecture more accurate. Also, measurements about the source code, such as lines of code for methods or structs are irrelevant. Secondly, clustering is a time consuming procedure and we need to minimize the combinations of relations that will be clustered to the absolute minimum. So we want to eliminate relations that give the same information with other relations.

For example, Fetch identifies three relations for type definitions and associations: a) Type Definition(TypeDef) where a type name is associated with a Class, a Struct or another Type Name that was previously defined, b) the information (HasType) about which Class or Struct is the type of a Function, Method, Attribute or Global Variable and c) a relation (HasTypeDef) that combines the previous ones and associates a Function, Method, Attribute or Global Variable with a type name. In the output file these relations could have the next form:

TypeDef FooClass fooType

HasType fooMethod FooClass

HasTypeDef fooMethod fooType

From these 3 relations keeping only the second one ensures that there is no information loss and that the minimal set of relations is kept. The other two relations are actually resolved and their information is included in the *HasType* relation. Keeping these extra relations would only increase the number of combinations of relations that need to be checked.

The 13 relations that we retained are presented and explained with examples.

1. **Calls** *function/method function/method*: Occurs when a function or a method calls another function or method.
   Example:

```
foo()
{
    ...
    bar();
    ...
}
```

   Created Relation:

   – Calls foo bar

2. **Include** *file file*: Appears when a file includes another file in C and C++ and represents the imports in Java.
   Example:

```
//in file foo.c

#include<foo.h>
...
```

   Created Relation:

   – Include foo.c foo.h

3. **Sets** *function/method attribute/global variable*: In a function or method an attribute, of a class or a struct, or a global variable is on the left side of an assignment statement.
   Example:

```
foo()
{
  globalVarA = 5;
}
```

   Created Relation:

   – Sets foo globalVarA

4. **Accesses** *function/method attribute/global variable*: In a method or a function, an attribute, of a class or a struct, or a global variable is on the right side of an assignment statement.
   Example:

```
foo()
{
  ...
  x = globalVarA;
  ...
}
```

   Created Relation:

   – Accesses foo globalVarA

5. **Class Belongs To File** *class/struct file*: Shows to which file a class or struct belongs.
   Example:

```
//in file Foo.java
class Foo{
  ...
}
```

   Created Relation:

   – ClassBelongsToFile Foo Foo.java

6. **Inherits From** *class class*: Represents the inheritance of classes in object-oriented programming.
   Example:

```
class Foo extends Bar {
   ...
}
```

   Created Relation:

   – InheritsFrom Foo Bar

7. **Has Type** *function/method/attribute/global variable class/struct*: As we explained before, this relation gives the information about the type of a function, method, attribute or global variable.
   Example:

```
public FooClass fooMethod() {
   ...
}
```

   Created Relation:

   – InheritsFrom Foo Bar

8. **Defined In** *function/method file*: Shows in which file a function or a method is defined. This is the file where the actual body of the function or the method is located. We should underline that for every function and method there is only one ”Defined in” relation in the output file of the extractor.
   Example:

```
//in file Foo.java
Foo bar()
{
   ...
}
```

   Created Relation:

   – DefinedIn bar Foo.java

9. **Declared In** *function/method file*: This relation is similar to the previous one, with one great difference. ”Declared In” shows in which file there is a declaration of the function or the method. As a result there can be more than one such relations for the same function or method in the output file of the extractor.
   Example:

```
//in file header.h
foo()
{
  ...
}

//in file bar.c
foo();
...
```

   Created Relations:

   – DefinedIn foo header.h
   – DeclaredIn foo header.h
   – DeclaredIn foo bar.c

10. **Attribute Belongs To Class** *attribute class/struct*: Shows to which class or struct an attribute belongs.
   Example:

```
class Foo
{
  attributeType bar = 0;
}
```

   Created Relation:

   – AttrubuteBelongsToClass bar Foo

11. **Accessible Entity Belongs To File** *global variable file*: This relation has the information to which file a global variable belongs.
   Example:

```
//in file bar.c

type foo = 0;
...
```

   Created Relations:

   – AccessibleEntityBelongsToFile foo bar.c

12. **Method Belongs To Class** *method class*: This relation shows which methods belong to each class in object oriented source code files.
    Example:

```
//in file Foo.java
class Foo
{
  bar1()
  {
    ...
  }

  bar2()
  {
    ...
  }
}
```

    Created Relations:

    – MethodBelongsToClass bar1 Foo
    – MethodBelongsToClass bar2 Foo

13. **Uses Type** *function/method struct/class*: This relation occurs when a class or a struct, not a primitive type, is used in a method or a function respectively.
    Example:

```

class Foo
{
  ...
}

class Bar
{
  bar1()
  {
    ...
    Foo x = new Foo();
    ...
  }
  ...
}
```

    Created Relation:

    – UsesType bar1 Foo

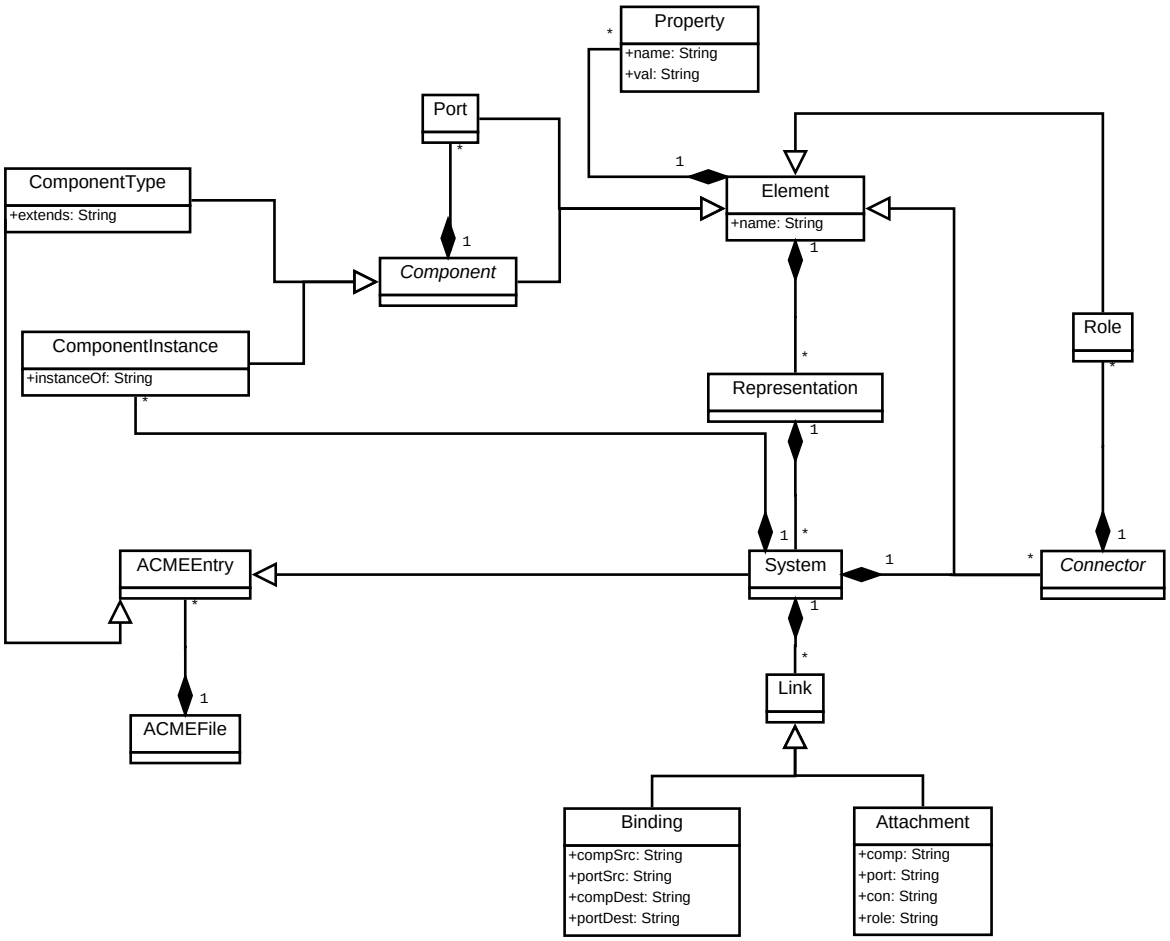

Figure 3.2: ACME Metamodel

3.3 Architecture Metamodel

In order to describe the architecture of a software system, a metamodel which describes architectures is required. Then, the population of this metamodel will create instances of the architecture of each system. The metamodel that was created was based on the metamodel of the architecture description language ACME [Garl97] which is presented on Figure 3.2. Figure 3.3 shows the metamodel that we created.

Table 3.1 describes the classes of the Modified Architecture Metamodel and Table 3.2 lists the types of entities that relate.

Generally, our metamodel, in agreement with the ACME metamodel, defines that each system contains elements which can either be *Components* or *Connectors*. An element, can have several *Properties* and depending on its type, the element contains different *Entities*.

To begin with, *Components* represent the generated clustered sets by the clustering procedure. Each *Component* has a unique name, which however does not have any information about the *Entities* that it contains. Each *Entity* that belongs to a *Component* can be an *Atomic Entity* (an *Entity* that does not enclose other *Entities*, and will only be connected to its parent *Entity*, e.g. attributes) or a *Composite Entity* (an *Entity* that contains other entities, e.g. files). All the *Entities* are characterized by their system-wide unique name. The *Entities* that are attached to a *Component* are organized in a spanning tree, based on the relations that connect them. For example, if there is a relation ”*Class Belongs To File* class file”, then the class would be the child of the file.

Class	Description
Component	Represents a cluster from a specific decomposition. It contains several *Entities*.
Connector	Represents a link between two *Components*. Each one contains one or more *Bindings*.
Binding	Specifies a link between two *Entities* that belong to two different *Components*. Represents one of the following relations: Call, Access, Set Variable, Include.
Property	Represents one of the following relations: Uses Type, Has Type, Inherits From, Defined In. It is attached to the Entity it refers to.
File	Refers to all the .c and .h files for procedural systems, and .java, .cpp and .h files for object oriented. Other *Entities* can be attached to *Files*.
Type	Represents classes in object oriented systems and structs in procedural systems.
Function	Represents functions in procedural systems and several *Properties* can be attached to it.
Method	Represents methods in object oriented systems. Public attributes and several *Properties* can be attached to it.
Global Variable	Represents global variables in procedural systems and is absent in object oriented systems.
Public Attribute	Represents public attributes of classes in object oriented systems and struct fields in procedural systems.

Table 3.1: Class Descriptions for Modified Architecture Metamodel

Element type	Type of children of element
System	Components, Connectors
Connector	Bindings
Module	Files
File	Types, Global Variables, Functions
Type	Methods, Inherits From, Has Type
Function	Has Type, Defined In, Uses Type
Method	Attribute, Has Type, Defined In, Uses Type

Table 3.2: The children of elements

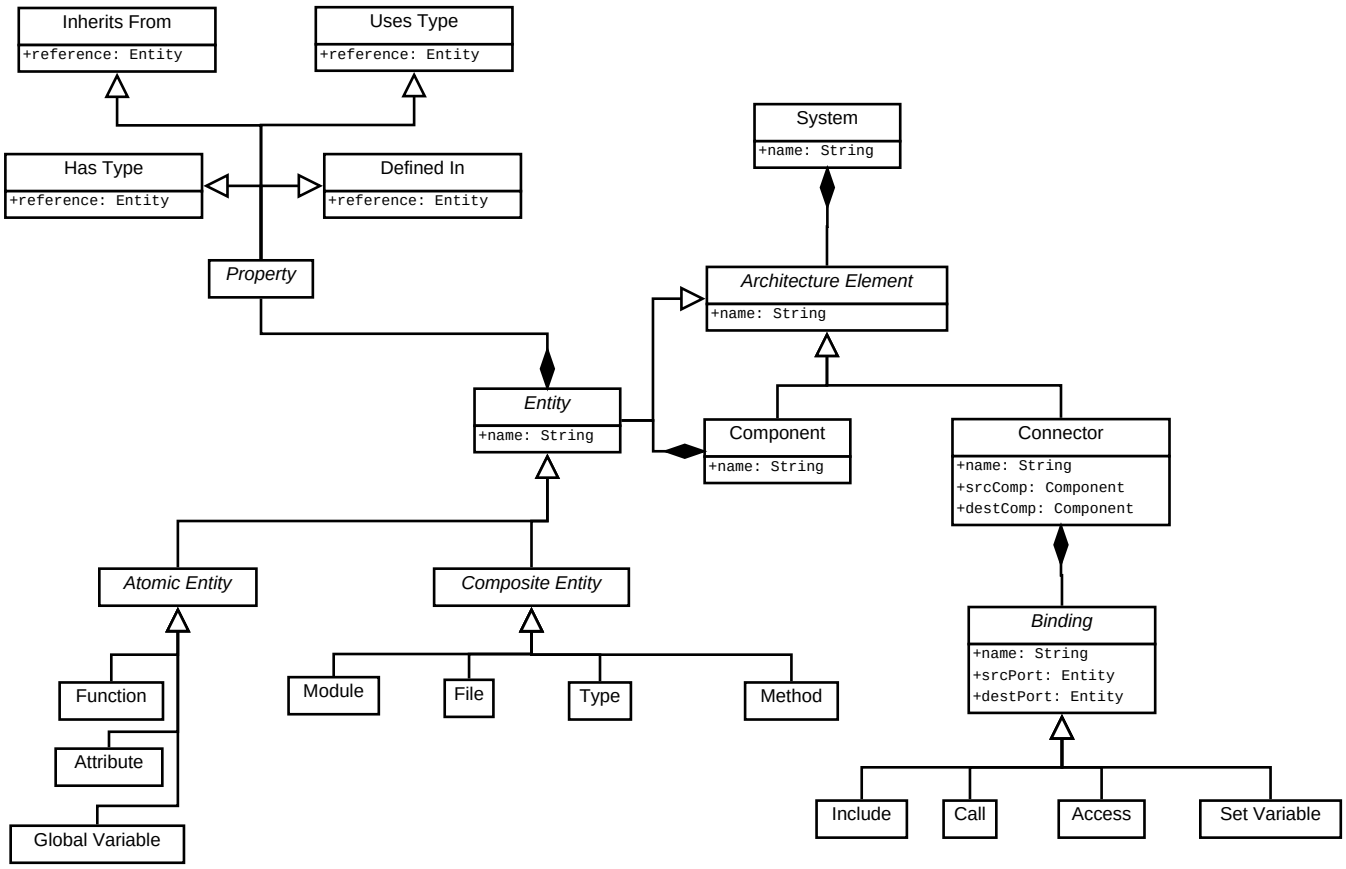

Figure 3.3: Modified Architecture Metamodel

Furthermore, *Entities* can have several additional *Properties* that characterize them. The type of a class, the files where a method or a function are declared and the class that another class inherits from are some of them. *Properties* are children of the *Entities* that they are related to, for example a class may have a *Property* about its type as a child.

On the other hand, there are *Connectors*. *Connectors* represent links between *Components*, however do not provide information about which specific *Entities* are connected. For this reason, each *Connector* contains one or more *Bindings*. *Bindings* connect specific *Entities* that belong to the *Components* that the *Connector* links. Each *Binding* is characterized by its type (Call, Access, Include, Set Variable) and the two *Entities* that the *Binding* refers to.

By populating this metamodel, we get instances of the system architecture. This procedure will be discussed in Section 5.2.

Chapter 4

Architecture Similarity

Given two instances of a software system, one of which is considered a gold standard, we needed to create a differencing algorithm that calculates the distance of the two instances by indicating the Entities that need to be moved, added and deleted in order for the one instance to be transformed to the gold standard.

Our algorithm assumes that one important condition is met, based on the metamodel that is used to create the instances. As we already mentioned, the names of the Entities are unique system-wide. As a result, if two Entities from two different instances are compared and their names are the same, we can safely assume that we are referring to the same Entity of the system. However, that is not applicable to Components, which are compared and matched with a different procedure that will be discussed in Section 4.1.

Our differencing algorithm is based on the UMLDiff algorithm [Xing05]. UMLDiff takes as input two versions of a system and retrieves the structural changes that occurred as the system evolved from the one version to the other. For UMLDiff the input is modelled as a a directed graph, where nodes are entities of the system and edges are the relations that connect the entities such as containment, declaration, method call etc.

Generally, UMLDiff traverses though the graphs of the two compared instances level by level, from the top to the bottom starting from the root of system. For each level, firstly the matched entities are identified. For two entities to be matched, their names need to be the same and they must also belong to the same level. After that, the entities that are renamed are identifies based on name and structure similarity.

- *Name Similarity* : calculates the similarity of the names of the entities by calculating the common adjacent character pairs that are contained in the two compared names.

- *Structure Similarity* : computes the similarity of two entities based on their connection with already matched entities. For example if two entities with similar names have as children other entities that have been identified as matched entities, then if the similarity score is higher than a set threshold the entities are identified as renamed.

Once the renamed entities have been identified, the next step is to identify which of the remaining entities have been moved to another part of the system. Same-named entities that belong to different levels and can not be characterized as matched, are here identified as moved entities. As soon as this step is completed, all the remaining entities of the initial version are identified as removed and all the remaining entities of the other version are identified as added entities. The output of UMLDiff is the labelled tree graph instance of the version of the system.

Based on UMLDiff we developed our differencing algorithm. It is a top-down algorithm that traverses from the root to to leaves of the tree graphs of two instances, one is the gold standard and the other is the instance generated for a specific combination of relations. We

identify matched, moved and deleted entities, but not renamed, since both instances come from the same version of the software system under examination, and not different versions like UMLDiff was examining.

The algorithm starts from the root level, moving on to Components, Composite Entities and finally Atomic Entities. At each level, except the Component level, based on the names of the entities which are unique, all entities are categorized as matched, moved and deleted. Components are matched differently, and we will discuss about it shortly.

Matched entities have the same name as well as belong to matched Components. Moved entities have the same name but belong to Components that are not matched. Finally deleted entities are present on the gold model instance but not on the other instance under examination. Once all entities have been identified, properties need to be matched.

Properties of matched entities are checked and if they have the same type and the attributes of the properties are the same, then these Properties are identified as matched. Otherwise, properties of the gold standard instance are categorized as deleted and the ones of the other instance as added.

Finally, Connectors are matched. Given two Connectors A and B, if the Components that Connector A joins have already been matched with the Components that Connector B joins, then A and B are identified as a match. On the other hand, if their Components are not a match, then Connector A is a deletion and Connector B is an addition.

As far as Bindings are concerned, if two Bindings belong to matched Connectors, have the same type and refer to the same entities, they are a match. On the contrary, Bindings that have different types or refer to different entities or belong to not matched Connectors are classified as deleted if they belong to the gold standard instance and as added if they belong to the other instance under examination.

4.1 Component Matching

As we discussed previously, Components are the sets that result from the clustering with some extra entities added during the population of the instance. However, since the names of the Components do not provide any information about their content we had to come up with a different way to match the Components of an instance to the Components of the gold standard.

To understand the way that Components were matched we will firstly discuss about the Hungarian Method [Kuhn55].

4.1.1 The Hungarian Method

General Problem

Given n workers and tasks, and an $n \times n$ matrix, W, containing the gain from assigning each worker to a task. The problem is how we can assign each worker x_i to a task y_j, such that the total gain is maximized over all possible assignments.

Given $X = \{x_1, \ldots, x_n\}, Y = \{y_1, \ldots, y_n\}$, and a matrix W where $W_{ij} = weight(x_i, y_i)$ is the weight of assigning x_i to y_i, find the matching assigning each x_i to each y_i such that the total weight is maximized.

Assuming that $\forall i, j \in \{1, \ldots, n\} : W_{ij} \geq 0$ this problem can be transformed to a Complete Weighted Bipartite Graph $G = (V, E)$:

- $V = X \cup Y$
- $E = \{(x_i, y_i)\}_{x_i \in X, y_i \in Y}$
- $weight(x_i, y_i) = W_{ij}$

The problem of finding an assignment that is a perfect matching is reduced to finding the perfect matching with maximum weight.

Definitions

1. A *labeling* for graph $G = (V, E)$ is a function $l : V \to R$, such that:

$$\forall (u, v) \exists \in E : l(u) + l(v) \geq weight(u, v) \tag{4.1}$$

2. An *Equality Subgraph* is a subgraph $G_l = (V, E_l) \subseteq G = (V, E)$, fixed on a labeling l, such that:

$$E_l = \{(u, v) \in E : l(u) + l(v) = weight(u, v)\} \tag{4.2}$$

The algorithm utilizes the Kuhn-Munkres Theorem: Given labeling l, if M is a perfect matching on G_l, then M is a maximal-weight matching of G. By the Kuhn-Munkres Theorem the problem of finding a maximum weight assignment is reduced to finding the right labeling function and any perfect matching on the corresponding equality subgraph.

Algorithm Idea

The main algorithm idea is to maintain both a matching M and an equality graph G_l, starting with $M = \emptyset$ and a valid l. In each step either augment M or improve the labeling $l \to l'$ until M becomes a perfect matching on G_l.

Augment the matching

Given labeling l, $G_l = (V, E_l)$, some matching M on G_l, and unmatched $u \in V, u \in M$:

1. A path is augmenting for M on G_l if it alternates between $E_l - M$ and M, and the first and last vertices of the path are unmatched in M. We keep track of an "almost" augmenting path starting at u.

2. If we can find an unmatched vertex v, then we create an augmenting path α from u to v.

3. Flip the matching by replacing the edges in M with the edges in the augmenting path that are in $E_l - M$.

4. Since we start and end unmatched, this increases the size of the matching, $|M'| > M$.

Improve the labeling

1. $S \subseteq X$ and $T \subseteq Y$ such that S, T represent the current "almost" augmenting alternating path between the matching M and outside other edges in $E_l - M$.

2. Let $N_l(S)$ be the neighbors to each node in S along E_l. $N_l(S) = \{v \mid \forall u \in S : (u, v) \in E_l$.

3. If $N_l(S) = T$ we cannot increase the alternating path and augment, so we must improve the labeling.

4. We compute $\delta_l = min_{u \in S, v \in T}\{l(u) + l(v) - weight(u, v)\}$

5. Improve $l \to l'$:

$$l'(r) = \begin{cases} l(r) - \delta_l & if r \in S \\ l(r) + \delta_l & if r \in T \\ l(r) & otherwise \end{cases} \tag{4.3}$$

   l' is a valid labeling and $E_l \subset E - l'$.

Algorithm

We start with some matching M and a valid labeling $l ::= \forall x \in X, y \in Y : l(y) = 0, l(x) = max_{y' \in Y}(weight(x, y'))$. Until M is a perfect matching:

1. Look for augmenting path

2. If augmenting path does not exist, improve $l \to l'$ and go to step 1.

The Hungarian Method, with small alterations, fits perfectly to solve our Component matching problem. The problem can be identified as a maximum weighted bipartite matching [Wils86]. The two partite sets are the Components of the instance under examination and the Components of the gold standard. The weight of the edges of the graph is the count of the same-named entities that the two Components that are connected have in common. We want to maximize the overall number of entities that are matched. Yet in order to solve this problem the two sets must have the same number of elements which is not always the case. To overcome this obstacle extra dummy Components are added to the set with the lowest cardinality and are connected with zero-weight edges to the other set. As a result the outcome of matching is not affected.

After that we have a matching between the Components of the two instances. Because of the nature of our problem, we can end up with unmatched Components –matched to the dummy Components that we added– which can lead to moved entities later on. However, our experiments showed that at least 80% of the Components of the instance that we are examining are matched with Components from the gold standard which is satisfactory.

Time complexity of the Hungarian Method is $O(n^3)$, nevertheless there are a lot of one-to-one matches between Components (Components that share same-named entities with only one Component of the other set), which improves running time by decreasing n and actually solving a smaller problem.

Listing 4.1 presents the algorithm for Component matching.

```
matchComponents(rA, rB, matchedComponents, added, deleted, next)
  set1 = rA.getComponents();
  set2 = rB.getComponents();
  graph = generateGraphNodes(set1, set2);
  for all e1 in set1
    for all e2 in set2
      createEdge(e1, e2, graph);
    next.addAll(e1.getChildren);
  matching = graph.HungarianMethod();
  matchedComponents.addAll(matching);
  for all e1 in set1 and not in matchedComponents
    added.add(e1);
  for all e2 in set2 and not in matchedComponents
    deleted.add(e2);
```

Listing 4.1: Component Matching

4.2 Entity and Property Matching

Listing 4.2 describes the algorithm that is used for identification of Entities and Properties. Given an Entity or Property *e1* from the instance under examination, algorithm 4.2 characterizes it as matched, moved or added. Based on its name, searchEntity() (line 2) retrieves the same-name Entity or Property respectively, *e2*, of the gold standard. If this exists and both *e1* and *e2* belong to the same logical level of the architecture then we have a match (lines 5-6). Otherwise, if they belong to different levels that is identified as a move (lines 8-9), else if *e2* does not exist at all, then *e1* is identified as an added Entity to the system (line 11). While this is done, every time a match or a move is found it is removed from *setB*, which contains the Entities and Properties of the initial system. As a results once all Entities and Properties have been identified, *setB* contains all the deleted Entities and Properties.

```
1  identifyEntity(e1, matched, moved, added, next, setB)
2    e2 = setB.searchEntity(e1.name);
3    if (e2 != null)
4      if (e2.level == e1.level)
5        matched.add(e1);
6        setB.remove(e1);
7      else
8        moved.add(e1);
9        setB.remove(e1);
10   else
11     added.add(e1);
```

Listing 4.2: Entity and Property Matching

4.3 Connector Matching

After Components, Entities and Properties have been identified, the next step is to match Connectors. We iterate through the Connectors of the instance under evaluation and try to find for each one a matching Connector from the gold standard. In order for two Connectors, *c1* and *c2*, to be considered a match, they need to have the same type (line 6) and additionally the Components that *c1* connects need to have already been matched to the Components that *c2* connects (line 8). The remaining Connectors are identified as added, if they initially belonged to the examined instance (line 12) and as deleted if they initially were part of the gold standard (line 13).

```
1  matchConnectors(rA, rB, matchedConnectors, added, deleted)
2    conA = rA.getConnectors();
3    conB = rB.getConnectors();
4    for all c1 in conA
5      for all c2 in conB
6        if (c1.type!=c2.type) continue;
7        else
8          if (isMatch(c1.components, c2.components))
9            matchedConnectors.add([c1, c2]);
10           conA.remove(c1);
11           conB.remove(c2);
12   added.addAll(conA);
13   deleted.addAll(conB);
```

Listing 4.3: Connector Matching

4.4 Binding Matching

Finally, only the Bindings are still unmatched. Matched Bindings share the same name as well as matched parent Connectors. Therefore, we iterate through all the Bindings of the instance we want to evaluate (line 2) and find same-named Bindings within the gold standard (lines 3-4). If they also have matched Connectors as parents they are identified as matched (lines 5-7), otherwise they are identified as moved (lines 9-10). If there is no such same-named Binding, then it is identified as a deleted Binding from the gold standard. Also, after each iteration, whenever a move or a match is identified, the Binding is removed from the set of Bindings of the gold standard and as a result all remaining Bindings in that set are identified as added Bindings to the gold standard.

```
1  matchBindings(bindsA, bindsB, matched, moved, added, deleted)
2    for all bind1 in bindsA
3      if bindsB.contains(bind1)
4        bind2 = bindB.get(bind1);
5        if bind1.getParent.isMatched(bind2.getParent)
6          matchedBindings(bind1);
7          bindsB.remove(bind2);
8        else
9          movedBindings.add(bind1);
10         bindsB.remove(bind2);
11     else
12       deleted.add(bind1);
13   added.addAll(bindsB);
```

Listing 4.4: Binding Matching

4.5 Differencing Score

The output of the previously explained algorithm is the distance score between the gold standard and the architecture instance under examination.

This score should fulfil certain criteria. Firstly, it should be stable, meaning that with small changes in the instance under examination, the score should change equally. In addition, once the score is normalized, the value 1 should represent completely difference instances, while 0 a complete match.

This score is calculated by 4.4.

$$diff_i = \frac{\begin{array}{r} AddedConnectors + DeletedConnectors + AddedComponents \\ + DeletedComponents + AddedEntities + DeletedEntities \\ + MovedEntities + AddedBindings + DeletedBindings \\ + MovedBindings \end{array}}{\begin{array}{r} TotalConnectors + TotalComponents + TotalEntities \\ + TotalProperties + TotalBindings \end{array}} \tag{4.4}$$

After these scores have been calculated for all the combinations of the system under analysis, they are normalized by scaling between 0 and 1 using 4.5.

$$normalized(diff_i) = \frac{diff_i - min(diff)}{max(diff) - min(diff)} \tag{4.5}$$

This value is sufficient to evaluate the relations of each system.

Chapter 5

Implementation of the Framework

5.1 Selection of the Clustering Methodology

In the context of this thesis, we had to choose which software algorithm to use. This choice was made based on two criteria respectively, execution speed and stability. Firstly, we wanted an algorithm that would be stable in the meaning that small changes in the input of the clustering should lead to small changes in the result of the clustering. Secondly, execution speed of the algorithm played an important role, since for our experiments the clustering algorithm would be executed about 50,000 times. By default clustering of large software systems is a time-consuming procedure, so we wanted to minimize the time of execution of this step since the results of the clustering will be used by the next steps of our work.

We compared two clustering algorithms, the ACDC algorithm 2.2.1 and the hill climbing clustering algorithm from the Bunch clustering suite 2.2.2, which for the rest of the thesis will be simply referred as Bunch.

Both algorithms presented advantages and disadvantages. As Wu et al.[Wu05b] have indicated the ACDC algorithm is more stable than Bunch, with the later even producing different clustering results for the same input. On the other hand, Bunch produces much more uniform clusters meaning that the sizes of the clusters do not present extremities (too big or too small clusters). But, since ACDC provided stable clusters, the fact that there were extremities in their sizes did not affect our experiments. However, Bunch had a disadvantage that we were not able to ignore. Execution speed of Bunch was significantly greater than ACDC as we presented before. Figures 5.1 and 5.2 compares the time that ACDC and Bunch required in order to cluster several procedural and object oriented systems.

For small systems, less than 60 KLOC, the difference between the clustering algorithms is small but significant, ACDC is 8.5 times faster than Bunch. However, in bigger systems containing 75 KLOC, ACDC is 56 times faster than Bunch, with the latter needing more than half an hour just for the clustering of one input file. When even larger systems are examined, like OpenSSL which contains almost 300 KLOC, ACDC requires about 15 minutes, which is one of the largest clustering execution times. On the other hand, Bunch requires 135 minutes and in order for the execution to be completed, we also had to increase the heap size of the program to 6GB.

We should also underline that execution time depends on the size of the source code of the system, but also depends on the number of different entities that there are as well as the way that these are connected. We observed that object oriented systems need slightly more time than same-sized procedural systems and this is why there are deviations in execution time between same-sized systems.

As a result, we choose the faster and more stable ACDC algorithm.

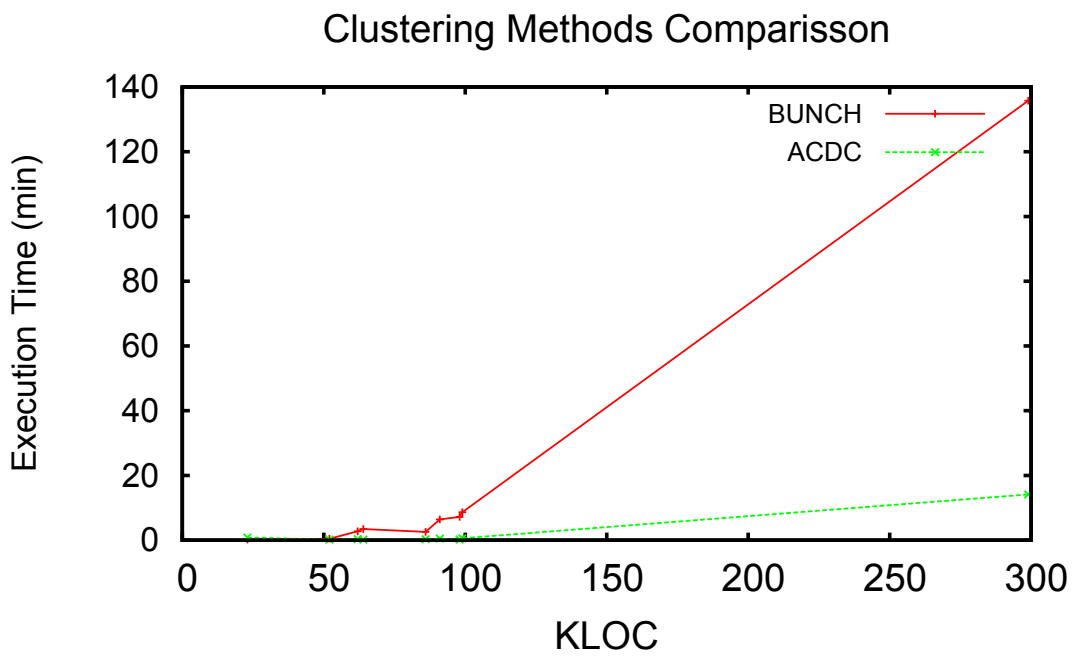

Figure 5.1: Comparison of Execution Time for ACDC and Bunch - Procedural Systems

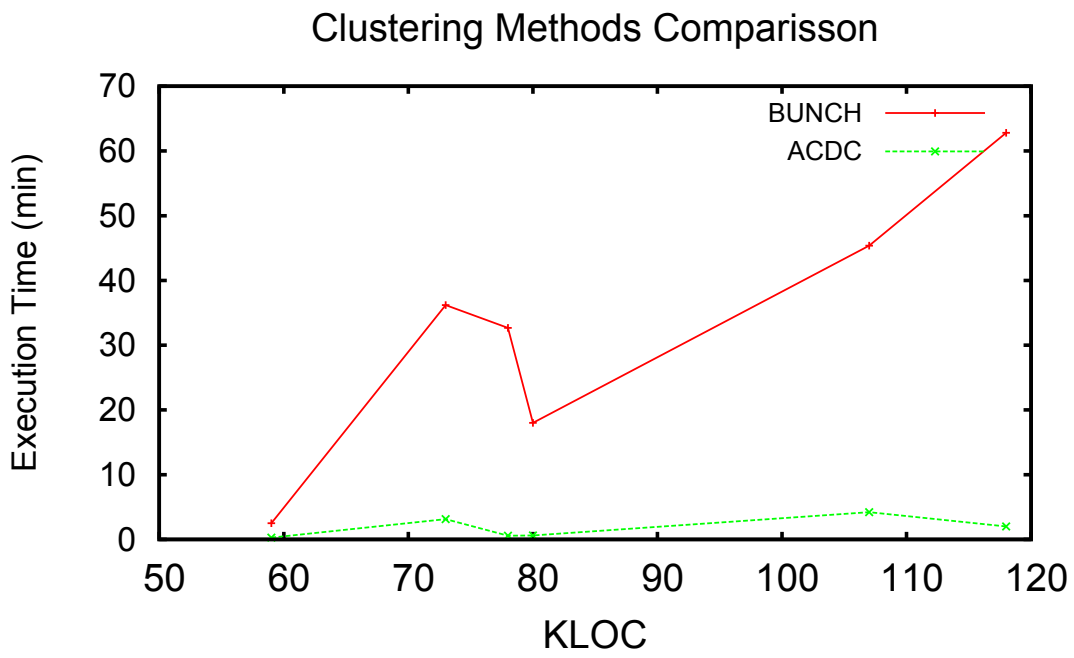

Figure 5.2: Comparison of Execution Time for ACDC and Bunch - Object Oriented Systems

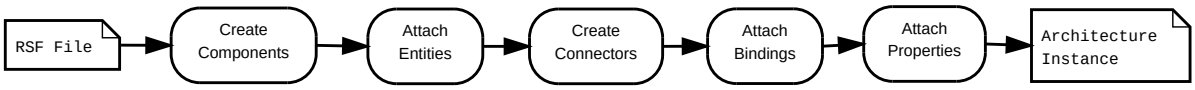

Figure 5.3: Steps for Population of Architectural Instances

5.2 Population of the Architecture Model

In order to be able to evaluate any software architecture, we first need to populate the meta-model and get an instance of the architecture that will be compared to the gold standard.

For each instance to be created we need:

1. the extracted relations from the source code of the software system
2. the result of the clustering procedure for the specific combination that is under examination

Each instance that is produced is a spanning tree. The steps of the population procedure are shown in Figure 5.3. It should be noted that the same procedure is followed for the population of both object-oriented and procedural instances.

We should emphasize that although clustering had as input only the relations that belong to a specific combination, and thus the output file contains only the Entities that are connected with these relations, during the population procedure we take into consideration all the Entities of the software system under examination. The reason for this is that we want to evaluate the impact of the relations of the combination to the system architecture and this is reflected on the results of the clustering. If only the Entities that were involved in the clustering were kept, the distance between architectures would be greater even though in reality the architectures are similar.

To begin with, the Components are created from the information of the clustering in order for the Entities that belong to each Component to be placed where they belong.

Secondly, the Entities need to be placed and we begin with the Files. From the clustering results we get the Component that each File belongs to and create an edge between the two nodes. Then, we move on to the next level, the Classes or Structs and Global Variables. For each one of them we get the Component that it belongs to and the File Entity that it is connected with. If the Component that the Entity belongs to is the same with the Component that the File belongs to, then and edge is created between the Entity and the File. Otherwise, and edge is created between the Entity and the Component that it belongs to. Before we move on to the next and final level of Entities, we have to also include in the instance the Classes or Structs and Global Variables that were not part of the clustering. These entities become children of the Files that they belong to or are declared in. Finally, we move on to the Attributes of the Classes or the Structs. If the Attribute and the Class or Struct belong to the same Component, an edge is created between the Attribute and the Class or the Struct, while if they belong to different Components, we simply attach the Attribute straight from the Component it belongs to.

If the relations that were taken into consideration during the clustering do not include any Files, then necessarily we move on the next level and therefore our graph will not have any Files. This happens because Files are top level entities that only belong to Components and if we do not obtain this information from the clustering, there is no way of certainly allocating Files to Components, like we do with the other types of Entities.

Once the previous step is completed, all the Entities, those that were part of the clustering procedure and the others that were not, have been positioned in the tree graph of the instance of the system architecture. The next step is to add the Connectors. Connectors represent the relations *Calls*, *Include*, *Accesses* and *Sets*. For a Connector to be made there is a restriction, the two Entities that interact with one of these relations have to be in different Components. Therefore, we go through all these relations that could create Connectors between Components. For each one, if the Entities of the relation belong to different Components and there is no Connector between them, a new Connector is created as well as a new Binding that includes information about the type of the relation and the Entities that are affected. The Binding becomes a child of the Connector that was just created and the Connector is connected with an edge to System root node. On the other hand, if a Connector already exists between two Components, only a Binding is created with the appropriate information and it is connected with the existing Connector.

Finally, we add the Properties to the Entities. The relations that describe properties are *Uses Type*, *Has Type*, *Inherits From* and *Defined In*. For each one of these relations in the relation file of the system, a new Property of the specified type with the appropriate information is created and connected with the Entity that the Property refers to. Once this is done for all the relations that describe properties, the population of the metamodel is done and the instance of the architecture of the software system is ready to be compared to the gold standard.

It should be noted that the instance that is considered as the gold standard is also created by this procedure when all the relations are taken into consideration during the clustering.

5.3 Population Examples

To make the population process more understandable, in this section we will present several simple examples.

The system that we will examine is not a real system and was created only for the purposes of this section and therefore is really small.

The initial RSF file of the system is presented on Listing 5.1.

```
Accesses M6 PA15
AttributeBelongsToClass PA15 T12
ClassBelongsToFile T10 F1
ClassBelongsToFile T11 F2
ClassBelongsToFile T12 F3
ClassBelongsToFile T13 F4
ClassBelongsToFile T14 F4
MethodBelongsToClass M5 T10
MethodBelongsToClass M6 T11
MethodBelongsToClass M7 T13
MethodBelongsToClass M8 T13
MethodBelongsToClass M9 T14
DefinedIn M7 F4
DeclaredIn M7 F4
DefinedIn M8 F4
DeclaredIn M8 F4
DefinedIn M9 F4
DeclaredIn M9 F4
Calls M5 M6
UsesType M6 T12
Include F2 F3
```

Listing 5.1: Complete RSF File

The result of the clustering of this system is presented below. There are 4 clusters containing the 15 Entities of the system.

```
SS(M5.ss) = M5, PA15, F3, M6, T12
SS(T10.ss) = F1, T10
SS(T11.ss) = F2, T11
SS(M9.ss) = M9, T14, F4, T13, M7, M8
```

Listing 5.2: Clustering Result

Combining the information presented above and using the populator that was described in Section 5.2 we can create an instance of the architecture of the system. Figure 5.4 illustrates the XMI that was created graphically.

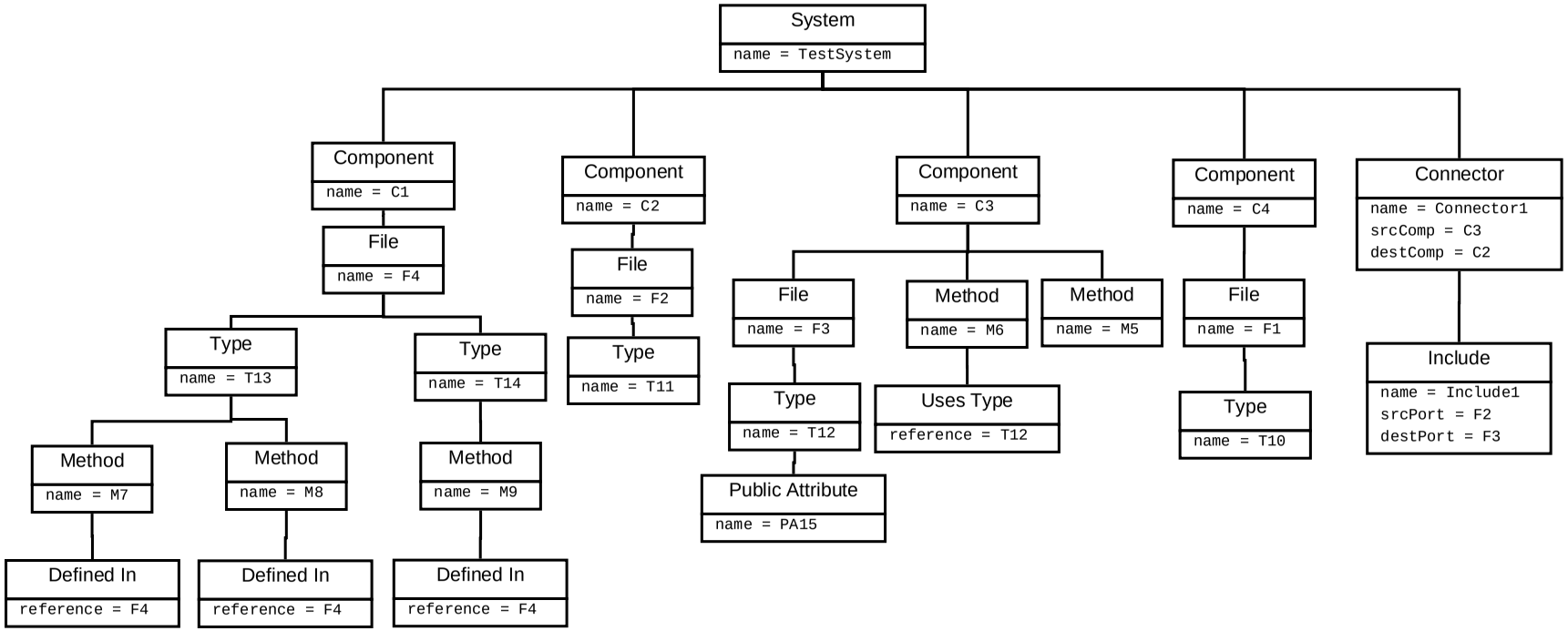

Figure 5.4: Populated Architecture Instance

Matched Elements	4
Added Elements	0
Deleted Elements	15
Moved Elements	0
Matched Clusters	1
Added Clusters	0
Deleted Clusters	3
Matched Connectors	0
Added Connectors	0
Deleted Connectors	1
Matched Bindings	0
Added Bindings	0
Deleted Bindings	1
Moved Bindings	0
Distance Score	0.789

Table 5.1: Result of Similarity Algorithm for Combination 1

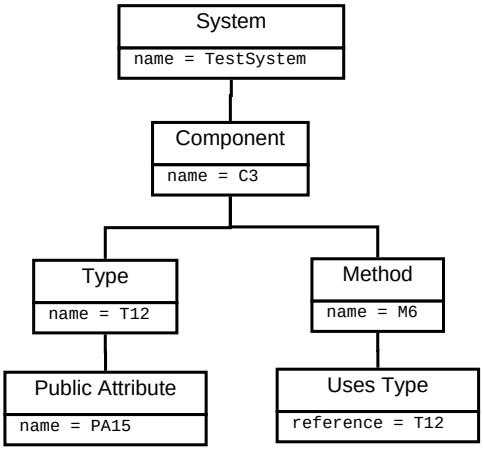

Figure 5.5: Populated Architecture Instance for Combination 1

To better understand the functionality of the populator, we will present the result of the population for some of the combinations of the relations. For each one we will present the combination of relations, the result of the clustering as well as the architecture instance that is created. In addition to population we will present the result of our similarity algorithm when comparing the several combinations to the architecture of the system 5.4.

1. Combination 1: UsesType

```
UsesType M6 T12
```

Listing 5.3: RSF File for Combination 1

```
SS(M6.ss) = M6, T12
```

Listing 5.4: Clustering Result for Combination 1

Matched Elements	11
Added Elements	0
Deleted Elements	5
Moved Elements	3
Matched Clusters	2
Added Clusters	1
Deleted Clusters	2
Matched Connectors	0
Added Connectors	1
Deleted Connectors	1
Matched Bindings	0
Added Bindings	1
Deleted Bindings	1
Moved Bindings	0
Distance Score	0.518

Table 5.2: Result of Similarity Algorithm for Combination 2

2. Combination 2: Method Belongs To Class, Defined In

```
MethodBelongsToClass M5 T10
MethodBelongsToClass M6 T11
MethodBelongsToClass M7 T13
MethodBelongsToClass M8 T13
MethodBelongsToClass M9 T14
DefinedIn M7 F4
DefinedIn M8 F4
DefinedIn M9 F4
```

Listing 5.5: RSF File for Combination 2

```
SS(M9.ss) = M9, T14, F4, M7, M8, T13
SS(M5.ss) = T10, M5
SS(M6.ss) = M6, T11
```

Listing 5.6: Clustering Result for Combination 2

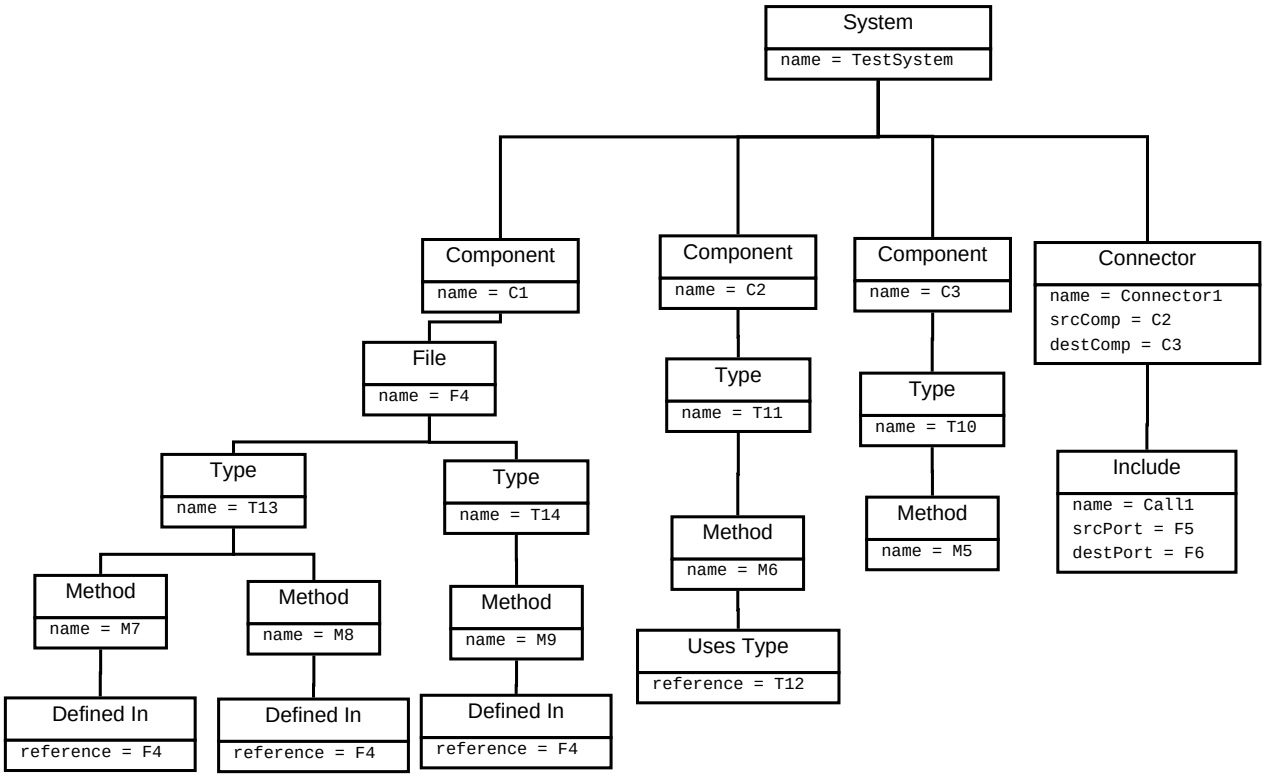

Figure 5.6: Populated Architecture Instance for Combination 2

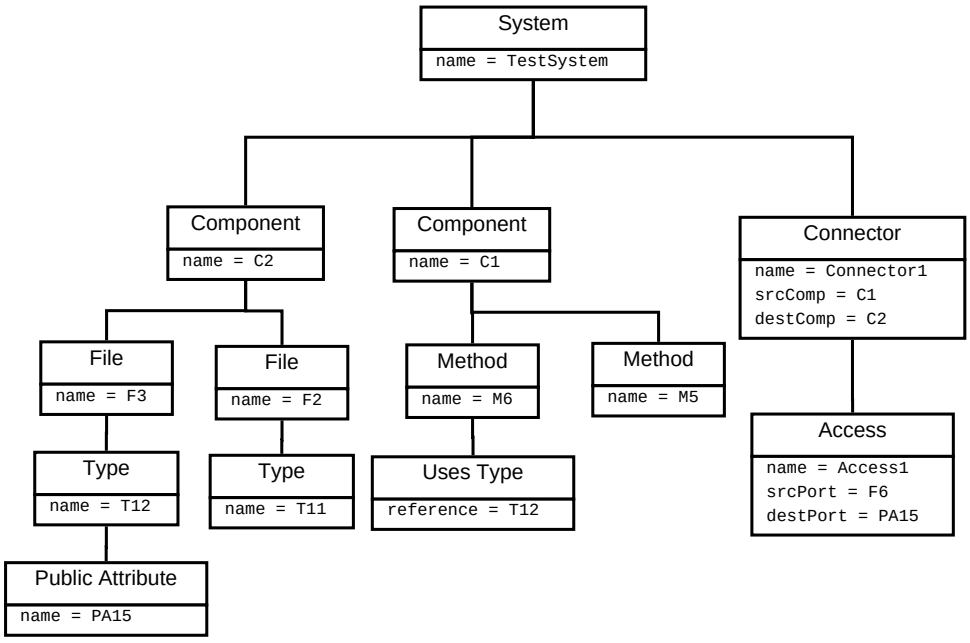

Figure 5.7: Populated Architecture Instance for Combination 3

Matched Elements	5
Added Elements	0
Deleted Elements	11
Moved Elements	3
Matched Clusters	2
Added Clusters	0
Deleted Clusters	2
Matched Connectors	1
Added Connectors	0
Deleted Connectors	0
Matched Bindings	0
Added Bindings	1
Deleted Bindings	1
Moved Bindings	0
Distance Score	0.657

Table 5.3: Result of Similarity Algorithm for Combination 3

3. Combination 3: Calls, Include

```
Calls M5 M6
Include F2 F3
```

Listing 5.7: RSF File for Combination 3

```
SS(M5.ss) = M6, M5
SS(F2.ss) = F2, F3
```

Listing 5.8: Clustering Result

Matched Elements	16
Added Elements	0
Deleted Elements	0
Moved Elements	3
Matched Clusters	4
Matched Tested Clusters	4
Matched Goal Clusters	4
Added Clusters	1
Deleted Clusters	0
Matched Connectors	0
Added Connectors	2
Deleted Connectors	1
Matched Bindings	0
Added Bindings	1
Deleted Bindings	0
Moved Bindings	1
Distance Score	0.657

Table 5.4: Result of Similarity Algorithm for Combination 4

4. Combination 4: Class Belongs To File, Calls, Declared In

```
ClassBelongsToFile T10 F1
ClassBelongsToFile T11 F2
ClassBelongsToFile T12 F3
ClassBelongsToFile T13 F4
ClassBelongsToFile T14 F4
DeclaredIn M7 F4
DeclaredIn M8 F4
DeclaredIn M9 F4
Calls M5 M6
```

Listing 5.9: RSF File for Combination 4

```
SS(T11.ss) = T11, F2
SS(T10.ss) = F1, T10
SS(M5.ss) = M5, M6
SS(F4.ss) = T13, F4, T14, M7, M8, M9
SS(T12.ss) = F3, T12
```

Listing 5.10: Clustering Result

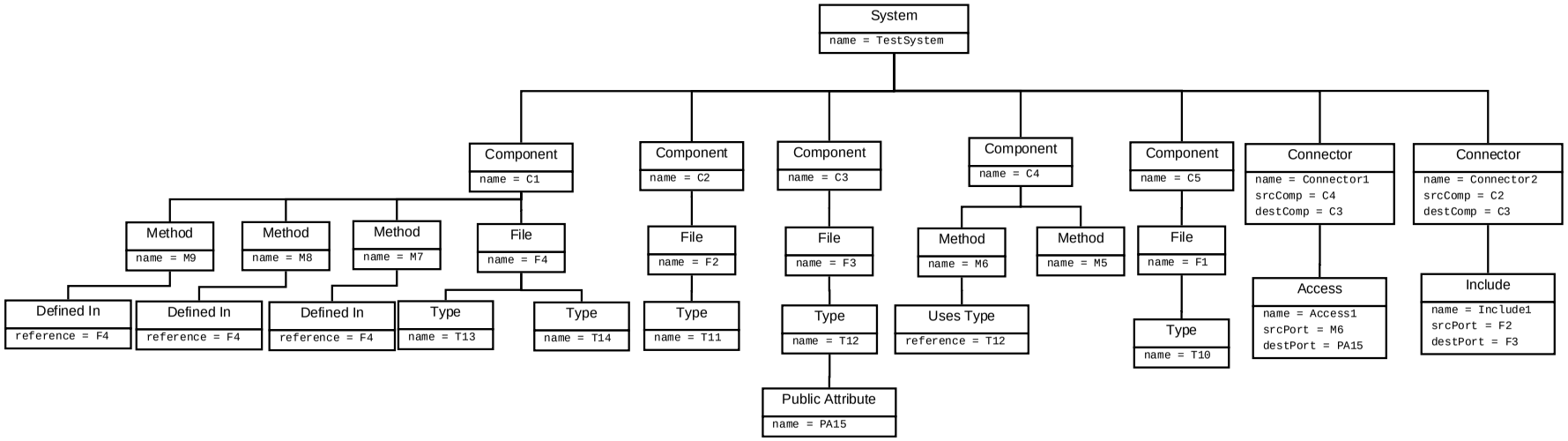

Figure 5.8: Populated Architecture Instance for Combination 4

Matched Elements	17
Added Elements	0
Deleted Elements	0
Moved Elements	2
Matched Clusters	4
Added Clusters	1
Deleted Clusters	0
Matched Connectors	0
Added Connectors	1
Deleted Connectors	1
Matched Bindings	0
Added Bindings	0
Deleted Bindings	0
Moved Bindings	1
Distance Score	0.096

Table 5.5: Result of Similarity Algorithm for Combination 5

5. Combination 5: Access, Calls, Class Belongs To File, Method Belongs To File

```
Accesses M6 PA15
ClassBelongsToFile T10 F1
ClassBelongsToFile T11 F2
ClassBelongsToFile T12 F3
ClassBelongsToFile T13 F4
ClassBelongsToFile T14 F4
MethodBelongsToClass M5 T10
MethodBelongsToClass M6 T11
MethodBelongsToClass M7 T13
MethodBelongsToClass M8 T13
MethodBelongsToClass M9 T14
Calls M5 M6
```

Listing 5.11: RSF File for Combination 5

```
SS(T10.ss) = T10, F1
SS(T11.ss) = F2, T11
SS(M5.ss) = M5, M6, PA15
SS(M9.ss) = M9, T14, F4, T13, M7, M8
SS(T12.ss) = F3, T12
```

Listing 5.12: Clustering Result

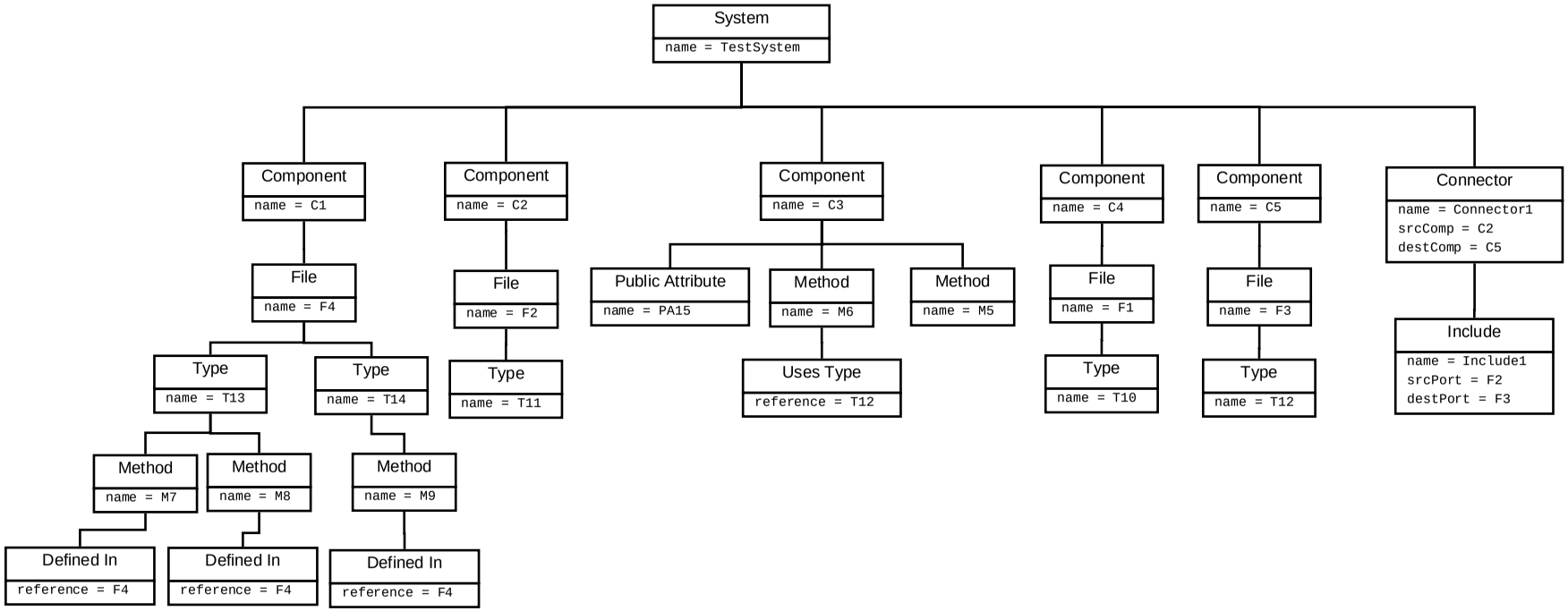

Figure 5.9: Populated Architecture Instance for Combination 5

Chapter 6

Experimentation Results

6.1 Preprocessing Results

6.1.1 Execution Times

In this section we will discuss the execution time of the several tools that we discussed.

Fetch

As far as execution time is concerned, the total time that Fetch requires to analyze several procedural and object oriented systems is presented on Figures 6.1 and 6.2 respectively. In addition to the graphs, because for smaller systems parsing times are not clearly visible, Tables 6.1 and 6.2 present the exact parsing times for procedural and object oriented systems respectively.

Generally in both procedural and object oriented systems, as the lines of code augment, parsing time rises as well. However, there is a difference between same sized procedural and object oriented systems. The first require less parsing time than the later. This happens because the analysis of the source code mainly depends on the interaction of the source code

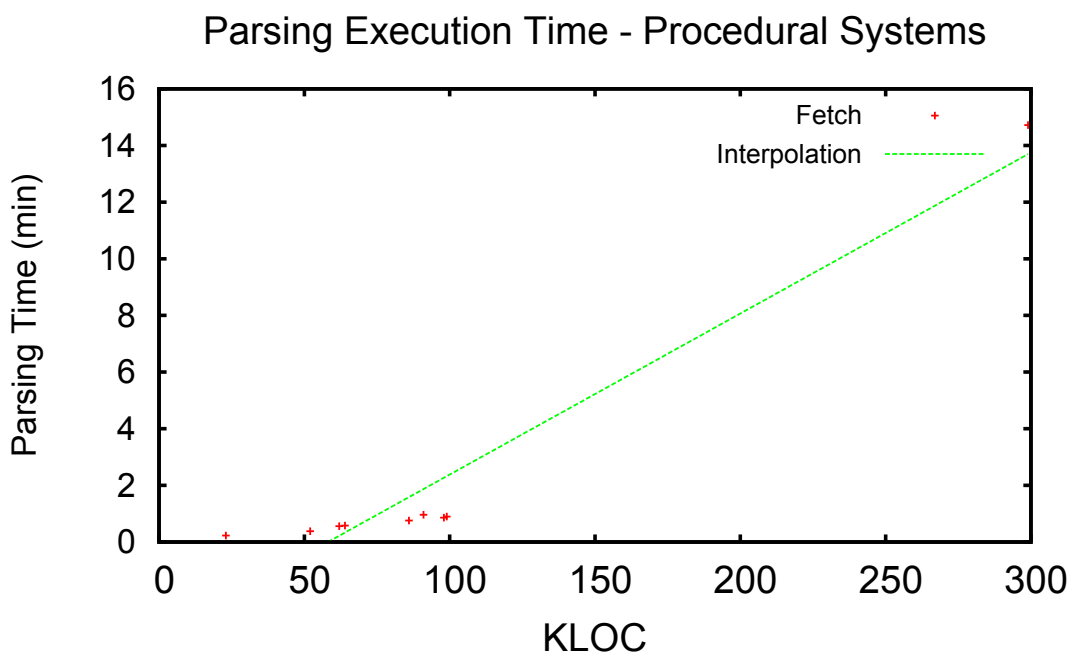

Figure 6.1: Fetch Execution Time for Procedural Systems

System	LOC	Parsing Time (min)
Freeglut	22832	0.23
Tcsh	52143	0.38
OpenVPN	61606	0.56
OpenSSH	63999	0.58
Putty	85716	0.76
Clips	91021	0.96
Zsh	98061	0.86
Bash	98871	0.90
OpenSSL	298767	14.72

Table 6.1: Fetch Execution Time for Procedural Systems

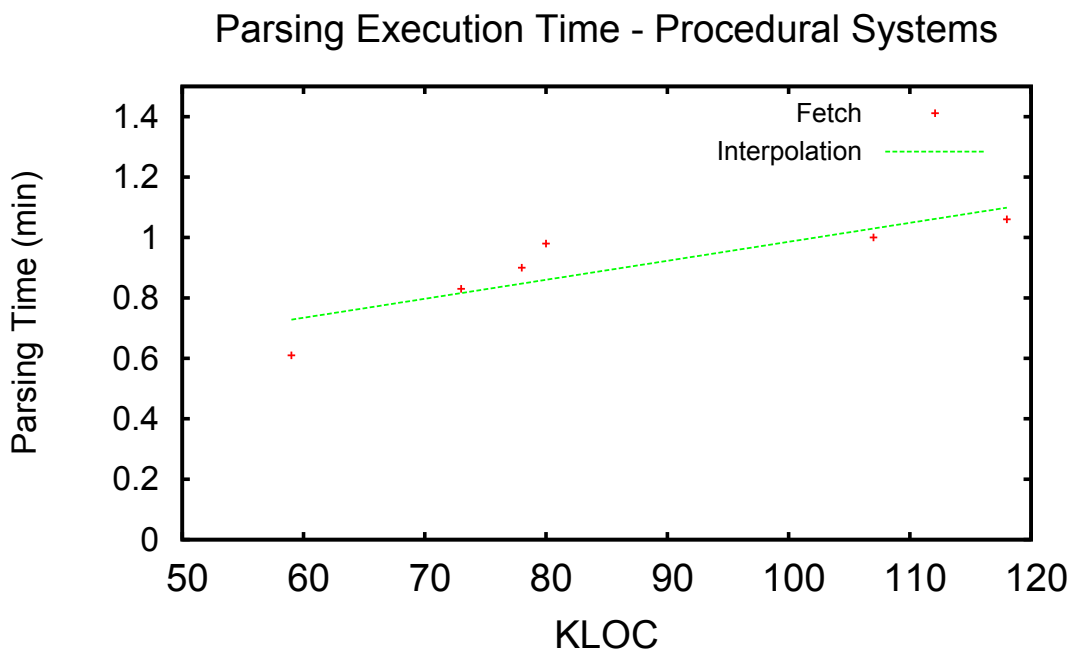

Figure 6.2: Fetch Execution Time for Object Oriented Systems

System	LOC	Parsing Time (min)
Texmaker	59434	0.61
Apache Ivy	72724	0.83
Apache Maven	78442	0.90
jHotDraw	80160	0.98
Apache Ant	107243	1.01
jEdit	118491	1.06

Table 6.2: Fetch Execution Time for Object Oriented Systems

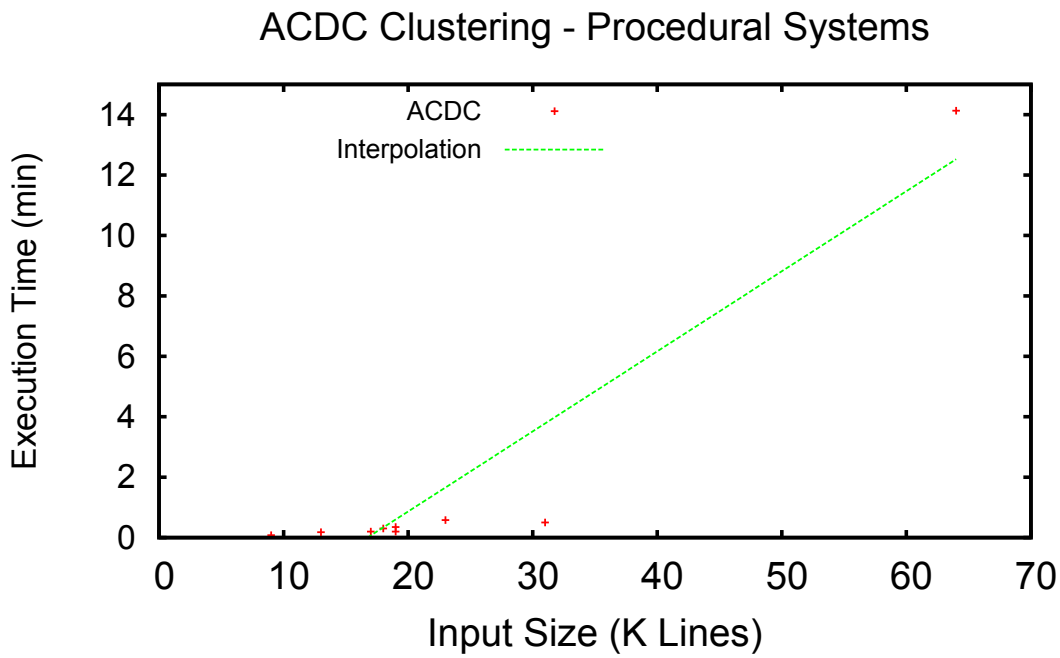

Figure 6.3: Execution Time for ACDC in comparison with input file size - Procedural Systems

System	Input File Size	No of Nodes	Time (Min)
Freeglut	9275	4161	0.80
Tcsh	12995	3138	0.18
Zsh	16905	5309	0.20
Putty	18439	6111	0.30
OpenVPN	18876	4411	0.35
OpenSSH	19370	4037	0.20
Bash	22850	7246	0.58
Clips	31217	6752	0.50
OpenSSL	62283	23385	14.13

Table 6.3: Execution time for ACDC for procedural systems

elements and not only on the size. In object oriented systems there are many cross-references to be resolved and there is a greater demand of queries to the database and therefore parsing time increases.

ACDC

ACDC algorithm takes as input an RSF file. Execution time depends on the size of the input file as well as the connections that exist between the entities of the system. We conducted experiments about the execution time of ACDC. Execution time varied from 10 seconds for the smaller systems to about 15 minutes for the larger ones. Figures 6.3 and 6.4 show the average clustering time for several procedural and object oriented systems respectively compared to the number of relations that were given as input. To make this more clear Tables 6.3 and 6.4 show the information about the procedural and object oriented systems respectively.

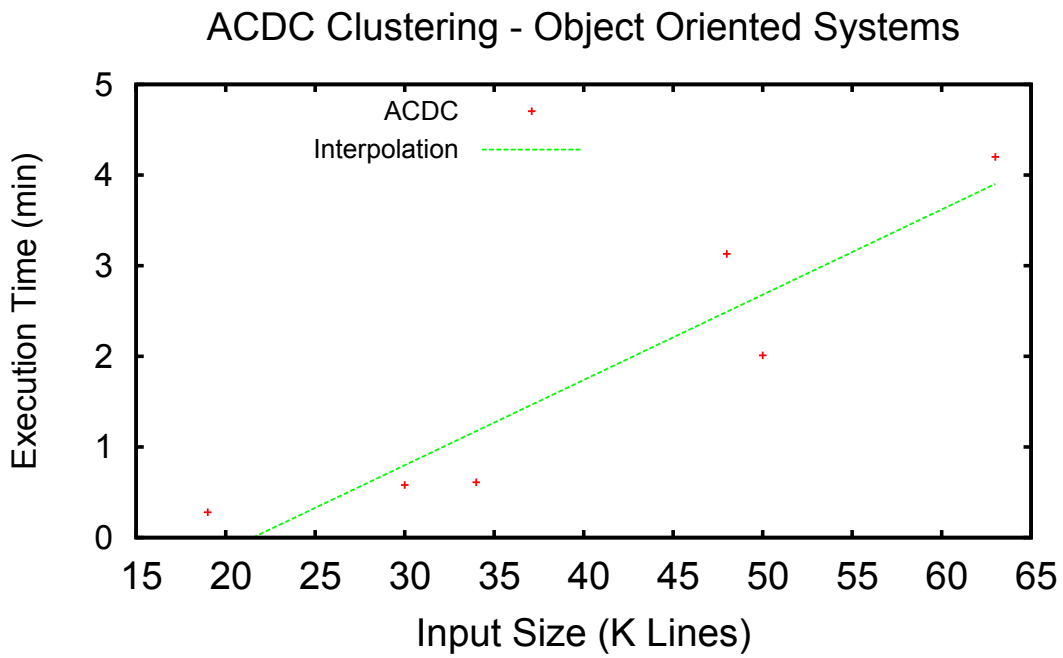

Figure 6.4: Execution Time for ACDC in comparison with input file size - Object Oriented Systems

System	Input File Size	No of Nodes	Time (Min)
TexMaker	18643	3970	0.28
Apache Maven	29827	11506	0.58
jHotDraw	33797	10712	0.61
Apache Ivy	48184	11280	3.13
jEdit	49903	12875	2.01
Apache Ant	63332	18149	4.20

Table 6.4: Execution time for ACDC for object oriented systems

We observe that object oriented systems need slightly more time than same sized procedural systems. This happens because even though we refer to same-sized input files, as we can see from Tables 6.3 and 6.4 procedural systems have almost twice the nodes object oriented systems have, and therefore more conflicts to be resolved during clustering.

Bunch

As far as execution time is concerned, Figures 6.5 and 6.6 show the average clustering time for several procedural and object oriented systems respectively compared to the number of relations that were given as input. Tables 6.5 and 6.6 show the information about the procedural and object oriented systems respectively. As we can see, Bunch is quite unstable and even for small input files time may vary from a couple of minutes to hours for the bigger files.

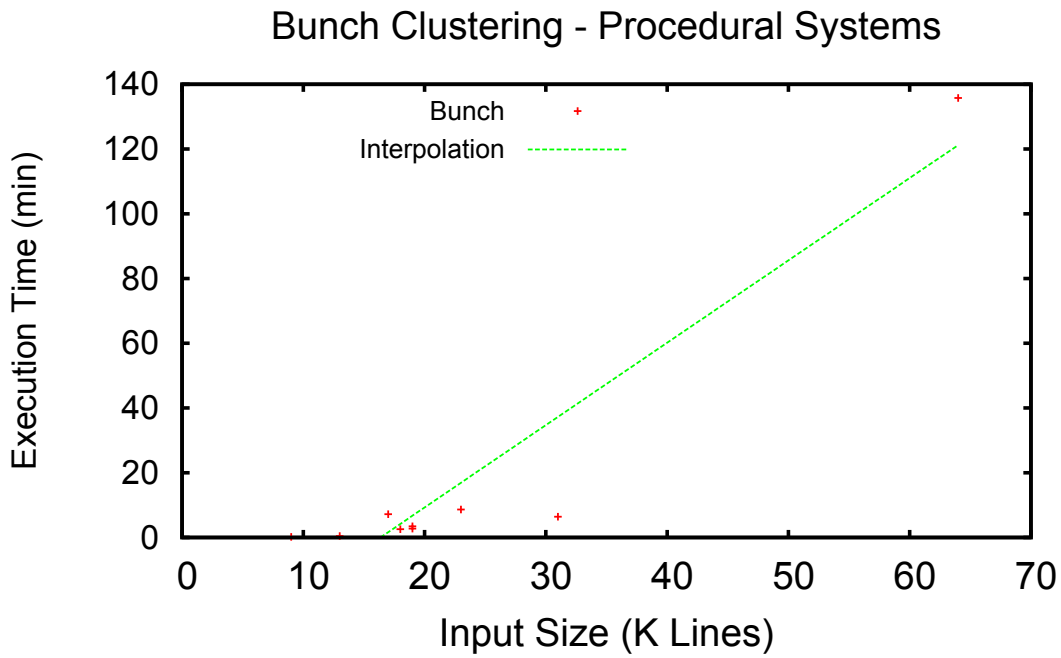

Figure 6.5: Execution Time for Bunch in comparison with input file size - Procedural Systems

System	Input File Size	No of Nodes	Time (Min)
Freeglut	9275	4161	0.20
Tcsh	12995	3138	0.46
Zsh	16905	5309	7.21
Putty	18439	6111	2.55
OpenVPN	18876	4411	2.73
OpenSSH	19370	4037	3.45
Bash	22850	7246	8.65
Clips	31217	6752	6.43
OpenSSL	62283	23385	135.78

Table 6.5: Execution time for Bunch for procedural systems

System	Input File Size	No of Nodes	Time (Min)
TexMaker	18643	3970	2.51
Apache Maven	29827	11506	32.68
jHotDraw	33797	10712	18.00
Apache Ivy	48184	11280	32.20
jEdit	49903	12875	62.80
Apache Ant	63332	18149	92.45

Table 6.6: Execution time for Bunch for object oriented systems

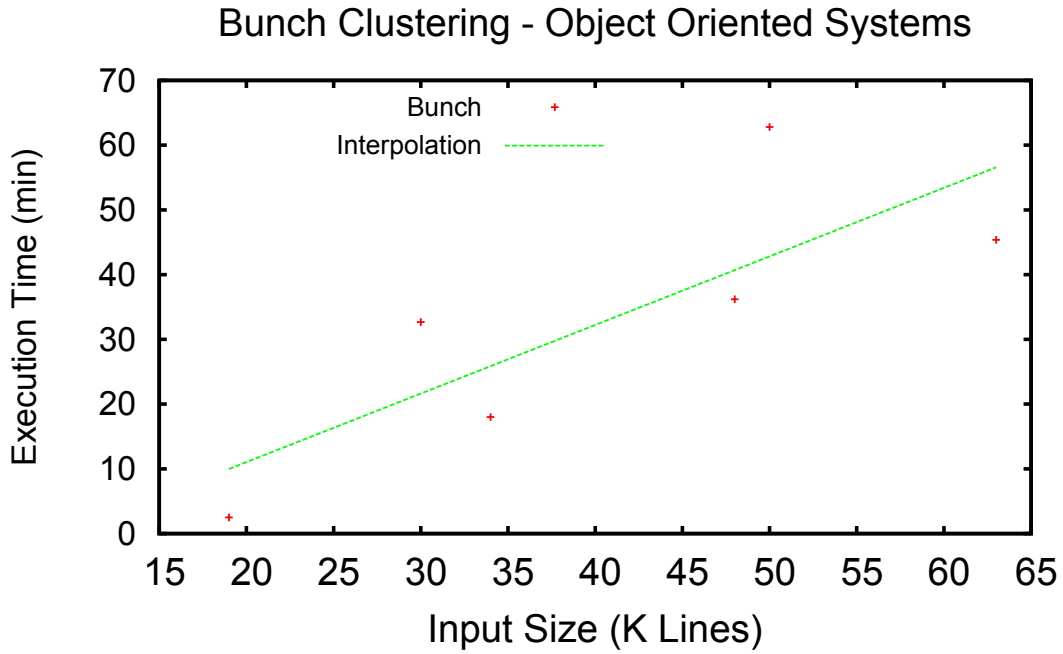

Figure 6.6: Execution Time for Bunch in comparison with input file size - Object Oriented Systems

6.2 Experimental Infrastructure

Before we go into detailed discussion about our experiments, we should first report the hardware and software configuration that was used. Because mainly clustering was an extremely time-consuming part of our work, we parallelized our experiments by using six quad-core processor machines. Processors are Intel Core i7 at 3,2 GHz with 8GB physical memory each. All machines run on Ubuntu Linux.

6.3 Systems under Analysis

In this section we present the systems that were analyzed during our experimentation process.

We examined several open source, large procedural and object oriented systems from various fields, such as Artificial Intelligence, Databases, Security etc.

Procedural Systems

The procedural systems that we examined consist of:

- *Bash* is a Unix shell and command language. It has been distributed widely as the shell for the GNU operating system and as a default shell on Linux and OS X. It is written in C.
- *Clips* is a software tool for building expert systems. It is the most widely used expert system tool. It is written in C.
- *OpenSSH* is a suite of security-related network-level utilities based on the SSH protocol. It is a free and open source alternative to the proprietary SSH. It is written in C.

System	LOC	RSF Lines	No of Nodes
Bash	99871	22850	7246
Clips	91021	31217	6752
Freeglut	22832	9275	4161
OpenSSH	63999	19370	4037
OpenSSL	298767	62283	23385
OpenVPN	61606	18876	4411
Putty	85716	18439	6111
Tcsh	52143	12995	3138
ZSH	98061	16905	5309

Table 6.7: Procedural Systems under Examination

- *OpenSSL* is an open source implementation of the SSL and TLS protocols. The core library implements basic cryptographic functions and utilities. It is written in C.
- *OpenVPN* is an open source software application that implements virtual private network techniques for creating secure point-to-point connections in routed configurations and remote access facilities. It is written in C.
- *Freeglut* is an open source alternative to the OpenGL Utility Toolkit (GLUT) library. It is written in C.
- *Putty* is a free and open source terminal emulator, serial console and network file transfer application. It supports several network protocols such as SCP, SSH, Telnet etc. It is written in C.
- *Tcsh* is a Unix Shell based on and compatible with the C shell (csh). It is written in C.
- *Zsh* is a Unix shell that can be used a powerful command interpreter for shell scripting. It is written in C.

Specific information about the procedural systems such as Line of Code (LOC), Number of relations included in the RSF File, Number of Nodes in the RSF File are presented in Table 6.7.

Object Oriented Systems

The object oriented systems that we examined consist of:

- *Apache Ant* is a software tool for automating software build processes. It is written in Java.
- *Apache Maven* is a software tool for automating software build processes used primarily for Java projects. It is written in Java.
- *Apache Ivy* is a sub-project of the Apache Ant project that is used to resolve project dependencies. It is written in Java.
- *jEdit* is a free and open source software text editor. It is written in Java.

System	LOC	RSF Lines	No of Nodes
Apache Ant	107243	63332	18149
Apache Ivy	72724	48184	11280
Apache Maven	78442	29827	11506
jEdit	118491	49903	12875
JHotDraw	80160	33797	10712
TexMaker	59434	18643	3970

Table 6.8: Object Oriented Systems under Examination

- *jHotDraw* is a Java framework for technical and structured Graphics. It is written in Java.
- *Texmaker* is a free and cross-platform LaTeX editor for Linux, OS X and Windows systems that integrates many tools needed to develop documents with LaTeX. It is written in C++.

Details about the object oriented systems are presented in Table 6.8.

6.4 Analysis Results

In this section we will present the results of our experiments.

First of all, Table 6.9 shows how easy or difficult it is to extract each relation from the source code. The extraction of a relation is considered easy if the source code does not need any special preparation to extract it, such as *Include* which can be obtained with a simple ”grep”. Furthermore, the extraction of a relation is considered of medium difficulty, if a small preprocessing is required, such as for the *Call* relation, which first needs to learn which are the function and then find the connection between them. Finally, relations that need the use of the linker or require information from the AST (Abstract Syntax Tree) are considered hard to extract. An example is the relations regarding types, which need to have the information about scopes, dependencies etc.

Relation	Easy	Medium	Hard
Access		√	
Accessible Entity Belongs to File	√		
Attribute Belongs to Class		√	
Class Belongs to File	√		
Call		√	
Declared In		√	
Defined In		√	
Has Type			√
Include	√		
Inherits From	√		
Method Belongs to Class	√		
Set Variable		√	
Uses Type			√

Table 6.9: Ease of extraction of relations from source code

6.4.1 Procedural Systems

Our results for the procedural systems are presented on the next figures and tables.

Figure 6.7 presents the average distance score for each combination from all the systems. As we can see, there are created three groups of points. Firstly, those above 0.6 who are considered bad combinations because the similarity between the two instances is very low. Then, we have the average combinations, with distance score between 0.3 and 0.6. Those combinations represent instances that are closer to the gold standard, but still are not sufficient. Finally, there are the combinations with distance score below 0.3. These combinations are very close to the gold standard and are considered good combinations that can be used for architectural extraction.

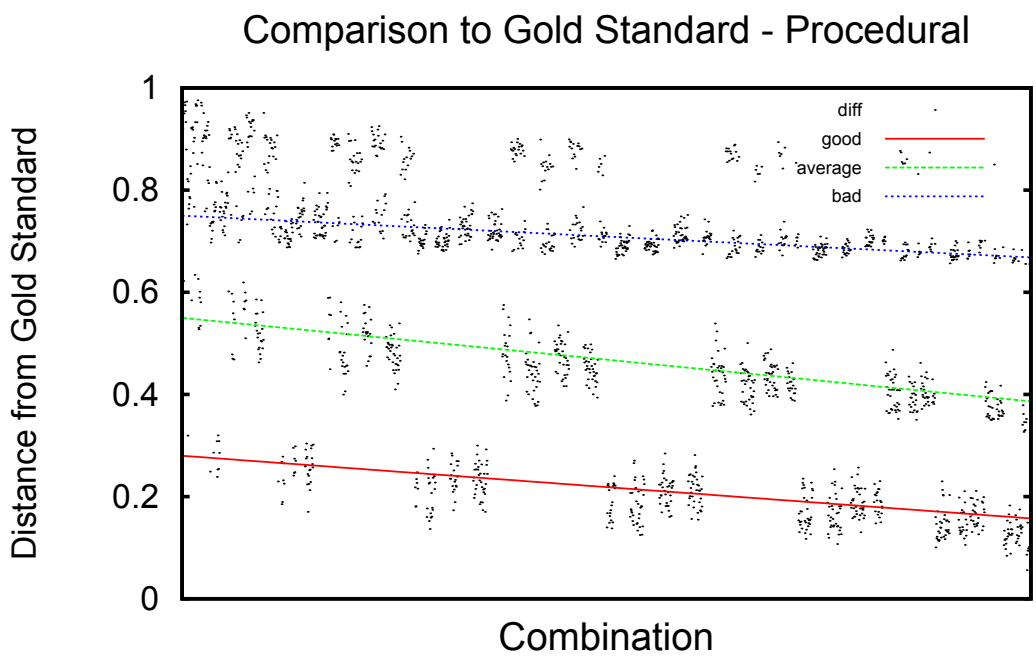

Figure 6.7: Average distance score for Procedural Systems

Figure 6.8 contains the same information as the previous figure (6.7), however we have sorted the data in descending order based on the distance score. Although in the x axis of this figure the combinations are in an unknown order to the reader and one might think it does not have physical meaning, it clearly shows something quite odd. There are two steps in the figure that are marked with the two vertical lines. The first line separates the combinations that do not have the relation *Call* (left) from those that have it (right). The second line separates the combinations that do not have the relations *Access* or *Call* (left) from those that have them both (right).

Table 6.10 shows the best combinations of relations. For each combination, we present the average Distance Score for all systems, the Z score as well as the p Value of a Chi Square Test. Z Score represents how many standard deviations the value is from the mean for this number of relations (per one, per two etc.). A Z score less than -2 represents the this value is in the top 0.5% of this set. Values that are so far from the mean, stand out of the set of combinations and are considered the best of them. As far as the p Value is concerned, we conducted simple Chi Square Tests in order to reveal the significance level of our results. Our Null Hypothesis was that each combination of relations and the outcome (good, average,

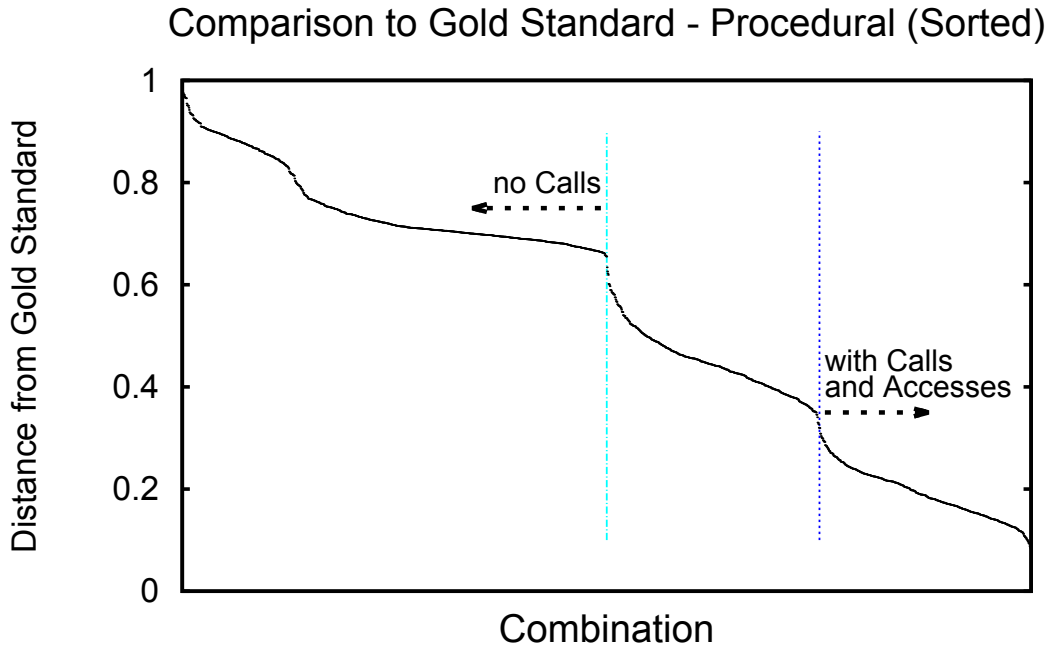

Figure 6.8: Sorted average distance score for Procedural Systems

bad) are independent. As we can see, for all the combinations that we present as best, the Null Hypothesis is rejected and with a probability of 100% we can conclude that there is a connection between the best combinations and the outcome.

On the other hand, Table 6.11 shows the worst relations that can be used for architectural extraction. As with Table 6.10, this one has information about the Distance Score, the Z Score as well as the p Value for a Chi Square Test of each combination. We can see here, that all these relations are independent of the outcome. And therefore, their use for architectural extraction is not suggested.

	Combination	Diff Score	Z Score < -2 p Value = 0
1	Call	0.622	-2.72
2	Call, Access	0.319	-3.43
	Call, Defined In	0.528	-2.00
3	Access, Call, Accessible Entity Belongs to File	0.249	-2.76
	Access, Call, Class Belongs to File	0.308	-2.42
	Access, Call, Declared In	0.238	-2.82
	Access, Call, Include	0.254	-2.73
	Access, Call, Has Type	0.319	-2.35
	Access, Call, Defined In	0.253	-2.73
	Access, Call, Set Variable	0.309	-2.42
	Access, Call, Attribute Belongs to Class	0.286	-2.54
	Access, Call, Uses Type	0.286	-2.55
4	Access, Call, Declare, Include	0.170	-2.46
	Access, Call, Accessible Entity Belongs to File, Declared In	0.185	-2.38
	Access, Call, Accessible Entity Belongs to File, Defined In	0.189	-2.36
5	Access, Call, Accessible Entity Belongs to File, Declared In, Include	0.137	-2.06

Table 6.10: Best combinations of relations - Procedural Systems

	Combination	Diff Score	Z Score	p Value
1	Attribute Belongs to Class	0.999	0.84	0.931
	Uses Type	0.973	0.60	0.969
	Declared In	0.971	0.58	0.954
	Defined In	0.966	0.53	0.954
	Has Type	0.955	0.42	0.969
2	Attribute Belongs to Class, Uses Type	0.976	1.08	0.941
	Class Belongs to File, Declared In	0.971	1.05	0.961
	Attribute Belongs to Class, Declared In	0.968	1.03	0.911
	Attribute Belongs to Class, Has Type	0.966	1.01	0.941
	Class Belongs to File, Defined In	0.966	1.01	0.961
3	Class Belongs to File, Declared In, Defined In	0.951	1.26	0.922
	Attribute Belongs to Class, Declared In, Defined In	0.951	1.26	0.828
	Attribute Belongs to Class, Class Belongs to File, Declared In	0.947	1.23	0.922
	Attribute Belongs to Class, Class Belongs to File, Defined In	0.942	1.21	0.922
	Attribute Belongs to Class, Has Type, Include	0.937	1.18	0.872
4	Attribute Belongs to Class, Declared In, Defined In, Uses Type	0.925	1.41	0.774
	Attribute Belongs to Class, Class Belongs to File, Declared In, Defined In	0.923	1.40	0.847
	Attribute Belongs to Class, Declared In, Defined In, Has Type	0.918	1.37	0.774
	Attribute Belongs to Class, Class Belongs to File, Declared In, Has Type	0.915	1.36	0.749
	Accessible Entity Belongs to File, Class Belongs to File, Declared In, Defined In	0.910	1.33	0.676
5	Accessible Entity Belongs to File, Attribute Belongs to Class, Declared In, Defined In, Include	0.900	1.54	0.399
	Attribute Belongs to Class, Class Belongs to File, Declared In, Defined In, Has Type	0.899	1.53	0.721
	Accessible Entity Belongs to File,Class Belongs to File, Declared In, Defined In, Include	0.899	1.53	0.663
	Attribute Belongs to Class, Class Belongs to File, Declared In, Defined In, Uses Type	0.896	1.51	0.840
	Accessible Entity Belongs to File, Attribute Belongs to Class, Class Belongs to File, Declared In, Has Type	0.894	1.51	0.840

Table 6.11: Worst combinations of relations - Procedural Systems

	Combination	Mean Diff Score with combination	SD	Mean Diff Score without combination	SD
1	Call	0.322	0.162	0.749	0.135
	Access	0.445	0.276	0.625	0.208
	Set Variable	0.502	0.238	0.568	0.277
2	Call, Access	0.189	0.094	0.796	0.139
	Call, Declared In	0.299	0.160	0.759	0.133
	Call, Defined In	0.299	0.159	0.759	0.134
3	Call, Access, Accessible Entity Belongs to File	0.166	0.080	0.811	0.123
	Call, Access, Declared In	0.166	0.093	0.806	0.136
	Call, Access, Defined In	0.167	0.093	0.807	0.136
4	Call, Access, Accessible Entity Belongs To File, Declared In	0.139	0.082	0.822	0.119
	Call, Access, Accessible Entity Belongs To File, Defined In	0.142	0.082	0.824	0.118
	Call, Access, Declared In, Include	0.147	0.083	0.816	0.130
5	Call, Access, Accessible Entity Belongs to File, Declared In, Uses Type	0.122	0.096	0.831	0.122
	Call, Access, Accessible Entity Belongs to File, Declared In, Include	0.122	0.093	0.837	0.107
	Call, Access, Accessible Entity Belongs to File, Defined In, Uses Type	0.123	0.095	0.832	0.122

Table 6.12: Average and Standard Deviation with and without a combination of relations - Procedural Systems

Finally, Table 6.12, presents the mean distance score for all the relations that contain this combination, as well as for all the relations that do not contain it in addition to the corresponding standard deviation for the best combinations of relations. The fact that when a combination of relations is present, we have a low distance score and standard deviation, while when this combination is absent the distance score rises significantly, shows that this combination is important for architecture recovery and should not be absent.

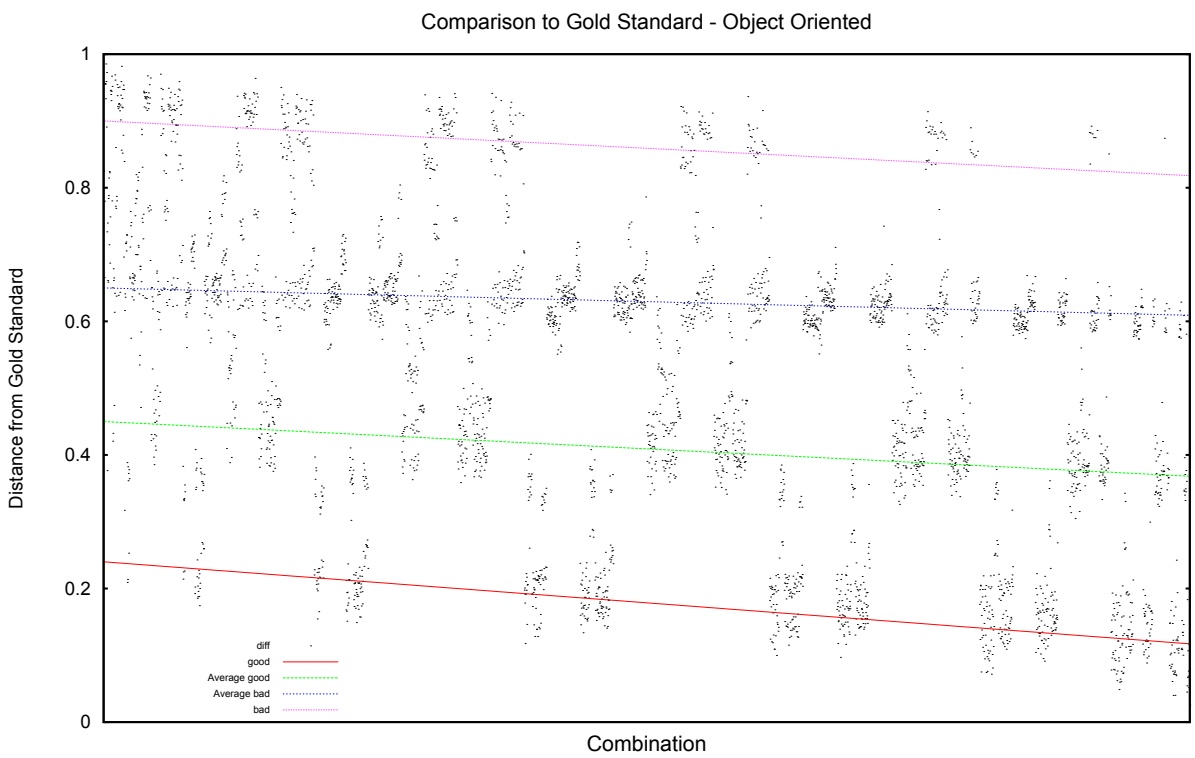

Figure 6.9: Average distance score for Object Oriented Systems

6.4.2 Object Oriented Systems

Our results for the object oriented systems are presented on the next figures and tables.

Figure 6.9 presents the average distance score for each combination from all the systems. As we can see, there are created four groups of points. Firstly, those above 0.8 who are considered bad combinations because the similarity between the two instances is extremely low. Then, we have the average bad combinations, with distance score between 0.8 and 0.6, which are still very different from the gold standard. Those combinations represent instances that are closer to the gold standard, but still are not sufficient. Furthermore, there are the combinations with distance score between 0.3 and 0.5 which are considered average good. Finally, there are the combinations with distance score below 0.3. These combinations are very close to the gold standard and are considered good combinations that can be used for architectural extraction.

Figure 6.10 contains the same information as the previous figure (6.9), however we have sorted the data in descending order based on the distance score. Although in the x axis of this figure the combinations are in an unknown order to the reader and one might think it does not have physical meaning, it clearly shows something quite odd. There are three steps in the figure that are marked with the three vertical lines. The first line separates the combinations that do not have the relations *Call*, *Access* or *Set Variable* (left) from those that have at least one of them (right). The second line separates the combinations that do not have the *Call* relation (left) from those that do (right). Finally, the third line separates the combinations that do not have the relations *Access* or *Call* (left) from those that have them both (right).

Table 6.13 shows the best combinations of relations for object oriented systems. For each combination, we present the average Distance Score for all systems, the Z score as well as the p Value of a Chi Square Test. Z Score represents how many standard deviations the

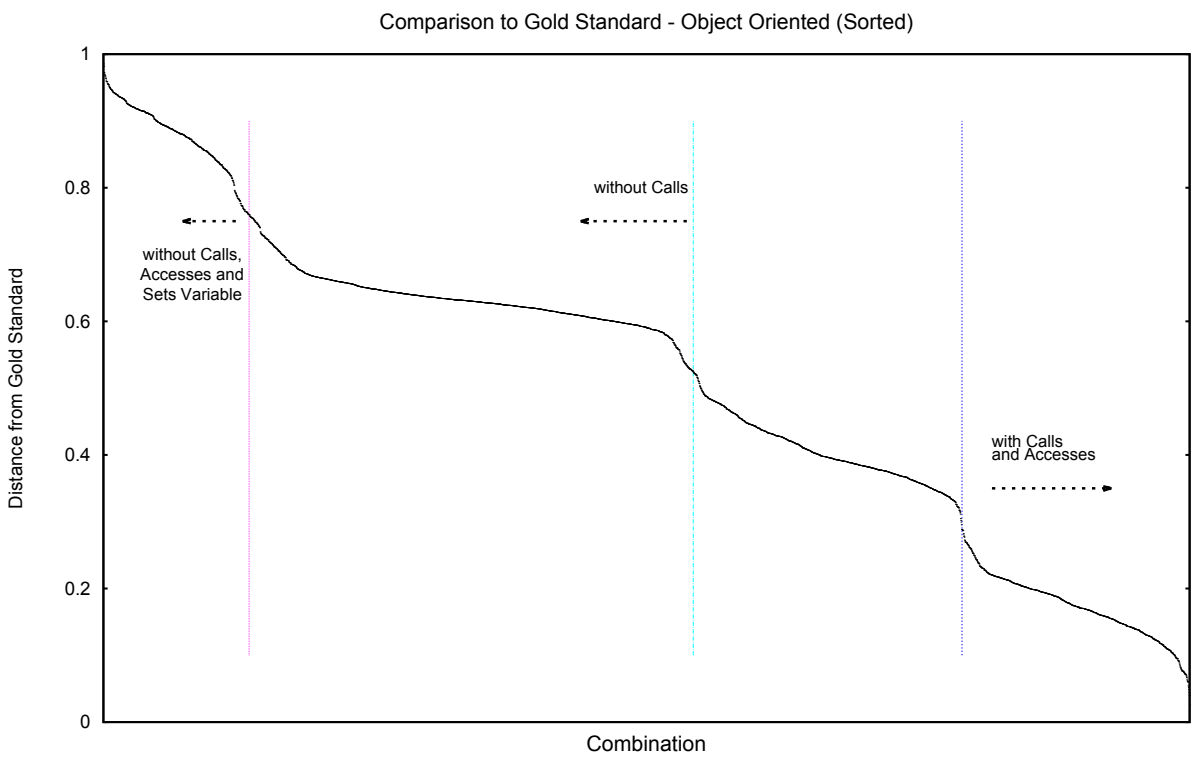

Figure 6.10: Sorted average distance score for Object Oriented Systems

value is from the mean for this number of relations (per one, per two etc.). A Z score less than -2 represents the this value is in the top 0.5% of this set. Values that are so far from the mean, stand out of the set of combinations and are considered the best of them. As far as the p Value is concerned, we conducted simple Chi Square Tests in order to reveal the significance level of our results. Our Null Hypothesis was that each combination of relations and the outcome (good, average good, average bad, bad) are independent. As we can see, for all the combinations that we present as best, the Null Hypothesis is rejected and with a probability of 100% we can conclude that there is a connection between the best combinations and the outcome.

On the other hand, Table 6.14 shows the worst relations that can be used for architectural extraction in object oriented systems. As with Table 6.13, this one has information about the Distance Score, the Z Score as well as the p Value for a Chi Square Test of each combination. We can see here, that all these relations are independent of the outcome. And therefore, their use for architectural extraction is not suggested.

Finally, Table 6.15, presents the mean distance score for all the relations that contain this combination, as well as for all the relations that do not contain it in addition to the corresponding standard deviation for the best combinations of relations. The fact that when a combination of relations is present, we have a low distance score and standard deviation, while when this combination is absent the distance score rises significantly, shows that this combination is important for architecture recovery and should not be absent.

	Combination	Diff Score	Z Score < -2 p Value = 0
1	Call	0.665	-2.43
2	Access, Call	0.377	-2.90
	Call, Defined In	0.421	-2.61
	Call, Declared In	0.432	-2.53
	Call, Method Belongs To Class	0.4741	-2.25
3	Access, Call, Declared In	0.209	-2.96
	Access, Call, Defined In	0.214	-2.93
	Access, Call, Method Belongs to Class	0.253	-2.71
	Access, Call, Attribute Belongs to Class	0.317	-2.34
	Access, Call, Include	0.361	-2.09
	Access, Call,Uses Type	0.364	-2.07
	Access, Call, Inherits From	0.372	-2.03
4	Access, Call, Defined In, Method Belongs to Class	0.174	-2.54
	Access, Call, Declared In, Include	0.185	-2.48
	Access, Call, Declared In, Method Belongs to Class	0.188	-2.47
	Access, Call, Defined In, Include	0.188	-2.47
	Access, Call, Declared In, Class Belongs to File	0.192	-2.44
	Access, Call, Declared In, Set Variable	0.196	-2.43
	Access, Call, Defined In, Set Variable	0.200	-2.40
5	Access, Call, Declared In, Include, Method Belongs to Class	0.149	-2.21
	Access, Call, Declared In, Class Belongs to File, Method Belongs to Class	0.151	-2.20
	Access, Call, Declared In, Attribute Belongs to Class, Method Belongs to Class	0.154	-2.18
	Access, Call, Defined In, Include, Method Belongs to Class	0.157	-2.17
	Access, Call, Defined In, Has Type, Method Belongs to Class	0.157	-2.17
	Access, Call, Declared In, Attribute Belongs to Class, Include	0.165	-2.13
	Access, Call, Declared In, Class Belongs to File, Include	0.166	-2.12

Table 6.13: Best combinations of relations - Object Oriented Systems

	Combination	Diff Score	Z Score < -2 p Value > 0.8
1	Inherits From	0.985	0.843
	Uses Type	0.972	0.71
	Method Belongs to Class	0.959	0.57
	Attribute Belongs to Class	0.958	0.57
	Defined In	0.956	0.54
2	Has Type, Uses Type	0.982	1.13
	Class Belongs to File, Declared In	0.969	1.04
	Attribute Belongs to Class, Method Belongs to Class	0.967	1.03
	Class Belongs to File, Defined In	0.961	0.99
	Has Type, Method Belongs to Class	0.959	0.98
3	Class Belongs to File, Declared In, Defined In	0.970	1.38
	Attribute Belongs to Class, Method Belongs to Class, Has Type	0.968	1.36
	Attribute Belongs to Class, Method Belongs to Class, Uses Type	0.964	1.35
	Has Type, Inherits From, Uses Type	0.958	1.31
	Attribute Belongs to Class, Inherits From, Method Belongs to Class	0.954	1.29
4	Attribute Belongs to Class, Has Type, Method Belongs to Class, Uses Type	0.964	1.68
	Class Belongs to File, Declared In, Defined In, Include	0.950	1.61
	Attribute Belongs to Class Has Type, Include, Method Belongs to Class	0.943	1.56
	Attribute Belongs to Class, Declared In, Defined In, Inherits From	0.941	1.56
	Attribute Belongs to Class, Include, Method Belongs to Class, Uses Type	0.941	1.55
5	Class Belongs to File, Declared In, Defined In, Has Type, Uses Type	0.941	1.87
	Attribute Belongs to Class, Has Type, Include, Method Belongs to Class, Uses Type	0.941	1.87
	Atribute Belongs to Class, Class Belongs to File, Declared In, Defined In, Method Belongs to Class	0.939	1.858
	Class Belongs to File, Declared In, Has Type, Include, Uses Type	0.935	1.84
	Attribute Belongs to Class, Declared In, Defined In, Has Type, Inherits From	0.935	1.84

Table 6.14: Worst combinations of relations - Object Oriented Systems

	Combination	Mean Diff Score with combination	SD	Mean Diff Score without combination	SD
1	Call	0.324	0.146	0.702	0.114
	Access	0.421	0.229	0.605	0.192
	Set Variable	0.482	0.194	0.543	0.258
2	Call, Access	0.202	0.080	0.764	0.128
	Call, Declared In	0.284	0.131	0.710	0.111
	Call, Method Belongs to Class	0.286	0.134	0.719	0.112
	Call, Defined In	0.288	0.131	0.713	0.111
3	Call, Access, Declared In	0.161	0.041	0.768	0.120
	Call, Access, Defined In	0.164	0.042	0.770	0.122
	Call, Access, Method Belongs to Class	0.168	0.071	0.781	0.121
4	Call, Access, Declared In, Method Belong to Class	0.129	0.032	0.793	0.109
	Call, Access, Defined In, Method Belongs to Class, Defined In	0.133	0.032	0.795	0.109
	Call, Access, Class Belongs to File	0.148	0.044	0.804	0.125
5	Call, Access, Declared In, Attribute Belongs to Class, Method Belongs to Class	0.113	0.030	0.809	0.107
	Call, Access, Declared In, Class Belongs to File, Method Belongs to Class	0.114	0.033	0.825	0.115
	Call, Access, Declared In, Method Belongs to Class, Include	0.115	0.031	0.792	0.114

Table 6.15: Average and Standard Deviation with and without a combination of relations - Object Oriented Systems

6.5 Interpretation of the Results

In this section we will discuss what we learned form our experiments. For both procedural and object oriented systems we will answer several question that explain our findings.

6.5.1 Procedural Systems

- *Are there any relations that should not be absent no matter what the cost is?*

  In fact there are two relations that their presence or absence makes significant difference. First of all, the *Call* relation seems to be the most important relation in procedural systems. As we can see from Figure 6.7 there are three groups of distance score, which are very clearly separated. The difference between the the top group and the other two below is the presence of the *Call* relation. This means that if we choose not to include the *Call* relation in our set of relations that will be used to recall the architecture we cannot obtain a distance score less than 0.6, which in no way is considered satisfying. As a result is is safe to assume that the *Call* relation should not be absent, even though

it is not the easiest to extract.

This is also visible in Table 6.12. In the first row we present the mean and the standard deviation of the distance score of all the combinations that contain and do not contain the *Call* relation. When in the combinations the relation is present, the mean distance score is 0.322 with a standard deviation of 0.162, while when the relation is absent the mean distance score rises to 0.749 with a standard deviation of 0.135. There is a big distance between the two mean scores which is in fact the price that ones has to pay if it is decided not to use the *Call* relation. Even if all other relations are used, except *Call*, the distance score is 0.66, which is higher than the distance score containing at least *Call*. Therefore, our proposal is to start with the *Call* relation and build up with more relations if more accuracy is required and more resources can be spend in the extraction of more relations.

In addition to *Call*, another relation is recommended to be part of the set of relations that will be used is *Access*. This relation combined with *Call* make the perfect double, that as we see in Table 6.10 is present in all the best combinations and is absent from all the worst combinations in Table 6.11. Furthermore, the double *Call, Access* is what makes the difference between the average and good relations in Figure 6.7. All the combinations below 0.3, which are considered good combinations for recovery, contain both *Call* and *Access* relations.

As a result, our proposal is that *Call* and *Access* should not be absent no matter what their cost of extraction is. The combinations that have these relations have a mean of 0.189 with a standard deviation of 0.094 compared to 0.796 and 0.139 respectively for those that do not have them. Therefore, if only two relations are to be used these are definitely *Call* and *Access* and if there is room for more these should be part of the set.

- *Are there any relations that should not be used?*

  The answer is no. In our experiments we did not encounter any relation that when it was added to a combination negatively affected the updated distance score.

  However, further to what we discussed before, some combinations affect the clustering mainly in a way that whether they are present or not does not have significant difference. These relations are *Has Type*, *Set Variable* and *Class Belongs to File*. As we can see from Table 6.10, when these relations are added to the *Call, Access* combination the distance score is slightly changed. The same happens if we add these relations to any combination. Given the fact that these relations are considered slightly difficult to be extracted from the source code and their contribution is not significant, it is safe to propose that these relations can be omitted for the sake of faster results with no loss in accuracy.

  Additionally, if we take a look at Table 6.11, we see that combinations containing information about structural aspects of the system, such as *Attribute Belongs to Class*, *Class Belongs to File* and *Accessible Entity Belongs to File*, but not how all these entities are connected produce the worst output. For that reason, these relations should be avoided to be used on their own for recovery without a relation that defines connections between these entities such as *Call*, *Access* or *Declared In*.

- *How many relations do we need for accurate architectural recovery?*

  Of course the answer to this question depends on which relations we are willing to use first of all. For example, if for some reason the *Call* relation is not used, even if the other 10 relations are used the distance score is about 0.65 which is very high. For that reason, if we choose the right relations, we ended up that 6 relations are enough. More specifically, the use of *Call* and *Access* is not negotiable. Then, by also including *Declared In*, *DefinedIn*, *Include* and *Accessible Entity Belongs to File* the distance score drops to 0.124 which is very accurate compared to the number of relation that were extracted. For more than 6 relation, because the amount of the information that is provided is close to the total amount of information of the system, provided that *Call* and *Access* are included, the difference between the distance scores of the combinations are insignificant (in the range 0.09 - 0.15). As a result, for more than 6 relation, if *Call* and *Access* are included, the choice of the other relations does not really matter.

6.5.2 Object Oriented Systems

As far as object oriented systems are concerned, although our finding are quite similar to those of the procedural systems, there are some essential differences.

- We observed that in procedural systems the relations that actually make a difference in extracting their architecture mainly involve the relations that show interaction between the components of the system,such as *Call* and *Access*. Although, in object oriented systems these relations are still important, the architecture is revealed actually from these relations in combination with the relations about the organization of the source code. As we can observe in Table 6.15 in order to get an architecture close to the gold standard (distance score less than 0.2) in addition to relations about the interaction of components we need relations about the organization of the source code. The organization of the classes in object oriented systems is a satisfactory representation of the system's architecture if combined with one relation about the interaction of the system's elements.

- Additionally, Figure 6.10 has a difference compared to the corresponding figure for procedural systems. There is a large fall in the beginning of the diagram, that as we already mentioned, separates the combinations that do not contain any of the relations *Call*, *Access* and *Set Variable* from those that do. As a result, we can safely conclude that the absence of information about the interaction of the Components results to great deviations from the gold standard.

- *Are there any relations that should not be absent no matter what the cost is?*

  In order to acquire an architecture close to the gold standard we need at least three relations, *Call*, *Access* and a relation such as *Method Belongs to Class*, *Declared In* or *Defined In* that represent the structure of the system. With only these three relations the distance score is about 0.25, which is adequate for an estimation of the architecture. For an even more accurate representation we can combine the relations that we previously mentioned and get a score less than 0.2 if four relations are combined or 0.15 if five relations are combined.

  Additionally, the relations about the structure of the source code are easy to extract from the source code, according to Table 6.9 with a simple parsing of the source code and therefore do not require extra effort.

If we choose to ignore these relations, as Table 6.15 shows, the average distance score rises higher than 0.7 which actually means that the remaining relations are unusable for architectural extraction.

Therefore, our proposal for object oriented systems is to start with the combination *Call*, *Access* and one of the following *Declared In*, *Defined In* or *Method Belongs to Class* and build up with more relations if more accuracy is required and more resources can be spend in the extraction of more relations.

- *Are there any relations that offer unnecessary information concerning architectural extraction?*

  Some combinations affect the clustering mainly in a way that whether they are present or not does not have significant difference. These relations are *Has Type*, *Inherits From* and *Uses Type*. As we can see from Table 6.13, when these relations are added to the *Call, Access* combination the distance score is slightly changed. The same happens if we add these relations to any combination. Given the fact that these relations are considered slightly difficult to be extracted from the source code and their contribution is not significant, it is safe to propose that these relations can be omitted for the sake of faster results with no loss in accuracy.

  Additionally, if we take a look at Table 6.11, we see that combinations containing information about structural aspects and types of the system, such as *Attribute Belongs to Class*, *Class Belongs to File*, *Method Belongs to Class* and *Uses Type* but not how all these entities are connected, produce the worst output. For that reason, these relations should be avoided to be used on their own for recovery without a relation that defines connections between these entities such as *Call*, *Access* or *Set Variable*.

- *How many relations do we need for accurate architectural recovery?*

  Of course the answer to this question depends on which relations we are willing to use first of all. For example, if for some reason the *Call* relation is not used, even if the other 11 relations are used the distance score is about 0.7 which is very high. For that reason, if we choose the right relations, we ended up that 5 relations are enough. More specifically, the use of *Call*, *Access* and a relation or two about the structure of the source code is not negotiable. By including more relation such as *Method Belongs to Class*, *Attribute Belongs to Class* or *Class Belongs to File* the distance score drops lower that 0.1 which is very accurate compared to the number of relation that were extracted. For more than 5 relation, because the amount of the information that is provided is close to the total amount of information of the system, provided that both structural and relation about interaction of Components are included, the difference between the distance scores of the combinations are insignificant (in the range 0.07 - 0.15). As a result, for more than 5 relation, if information about the structural elements ana how these are connected are included, the choice of the other relations does not really matter.

Chapter 7

Conclusion and Future Work

We conclude this thesis with a summary of what was described and presented so far. Firstly, we will summarize the conclusions that where reached during the design, modeling and experimentation that we conducted. Then, this thesis will end with reference to the issues we felt require further investigation in the form of proposals for future research.

We designed and implemented a framework in order to evaluate the effect that source code extracted relations have on cluster based architectural extraction. We went a step further from what has been researched so far and our work enabled us to respond to the questions that we set in the introduction.

- As we observed, the selection of the relations has a huge impact on the outcome of the extraction. Even one relation can make a great change and therefore the selection of the relations that are used should be made with caution.

- Depending on the accuracy that we want, we can get an accurate architecture by using only a subset of relations. As we noticed, in procedural systems, relations about the interaction of the Components, such as the ”Call” and ”Access” relations, should never be omitted and then as the number of relations used approaches the total number of relations of the system, the extracted architecture becomes more accurate. In object oriented systems, the organization of the source code reveals much information about the architecture and therefore there is the need for relations about the structure of the source code in order to extract an accurate architecture.

- As we expected, the combinations that have the greater number of relations produce the closest architectures to the real one. However, to our big surprise, when the relations that we consider as the most important are missing, the similarity deteriorates dramatically, to the point that it is best to use only the two good relations to get an outcome than the other ones combined.

- Fortunately, there are no relations that deteriorate the outcome and should be avoided. However, we found certain relations that their presence or not does not make any significant difference and can be omitted for the sake of a faster but still accurate extraction.

- Finally, a very important observation that we made was that six relations for procedural systems and five for object oriented, if selected properly, give a very accurate outcome with relatively little effort. When architectural extraction is used in real systems and not for research reasons, the fact that instead of eleven or twelve relations, six or five are enough to get an accurate estimate of the architecture of the system has a great impact on the time required which is crucial.

We believe that our work set the base for further experimentation on this subject. However, there are several aspects that could be explored further.

We analyzed several systems written in C, C++ and Java, and it would be interesting to experiment on other popular programming languages, such as Python which supports multiple programming paradigms (object oriented, imperative, functional, procedural) and see if relations that we now consider important are then insignificant.

Also, larger systems should be examined and from different application domains to discover if the results remain the same or the domain affects the importance of the relations.

Finally, it would be really interesting to experiment on distributed systems and include in the relations that are evaluated the relations involving the messages that are interchanged by the connected nodes. Maybe these messages hide valuable information about the architecture of the system.

Bibliography

[Anqu99] Nicolas Anquetil, Cédric Fourrier and Timothy C. Lethbridge, "Experiments with Clustering As a Software Remodularization Method", in *Proceedings of the Sixth Working Conference on Reverse Engineering*, WCRE '99, pp. 235–, Washington, DC, USA, 1999, IEEE Computer Society.

[Bass13] Len Bass, Paul Clements and Rick Kazman, *Software Architecture in Practice*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2013.

[Baue04] Markus Bauer and Mircea Trifu, "Architecture-Aware Adaptive Clustering of OO Systems", in *8th European Conference on Software Maintenance and Reengineering (CSMR 2004), 24-26 March 2004, Tampere, Finland, Proceedings*, pp. 3–14, 2004.

[Bois07] Bart Du Bois, Bart Van Rompaey, Karel Meijfroidt and Eric Suijs, "Supporting Reengineering Scenarios with FETCH: an Experience Report", *ECEASST*, vol. 8, 2007.

[Cora11] Anna Corazza, Sergio Di Martino, Valerio Maggio and Giuseppe Scanniello, "Investigating the Use of Lexical Information for Software System Clustering.", in Tom Mens, Yiannis Kanellopoulos and Andreas Winter, editors, *CSMR*, pp. 35–44, IEEE Computer Society, 2011.

[Dean01] Thomas R. Dean, Andrew J. Malton and Richard C. Holt, "Union Schemas as a Basis for a C++ Extractor", in *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01, Stuttgart, Germany, October 2-5, 2001*, p. 59, 2001.

[DeBa94] Jean-Marc DeBaud, Bijith Moopen and Spencer Rugaber, "Domain Analysis and Reverse Engineering.", in Hausi A. Müller and Mari Georges, editors, *ICSM*, pp. 326–335, IEEE Computer Society, 1994.

[famo] "FAMOOS", http://scg.unibe.ch/archive/famoos/.

[Garc11] Joshua Garcia, Daniel Popescu, Chris Mattmann, Nenad Medvidovic and Yuanfang Cai, "Enhancing Architectural Recovery Using Concerns", in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pp. 552–555, Washington, DC, USA, 2011, IEEE Computer Society.

[Garc13] Joshua Garcia, Igor Ivkovic and Nenad Medvidovic, "A comparative analysis of software architecture recovery techniques", in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pp. 486–496, 2013.

[Garl97] David Garlan, Robert T. Monroe and David Wile, "Acme: An Architecture Description Interchange Language", in *Proceedings of CASCON'97*, pp. 169–183, Toronto, Ontario, November 1997.

[Imbe91] Mike Imber, "Software Engineering Environments", 1991.

[Kosc00] Rainer Koschke and Thomas Eisenbarth, "A Framework for Experimental Evaluation of Clustering Techniques", in *8th International Workshop on Program Comprehension (IWPC 2000), 10-11 June 2000, Limerick, Ireland*, pp. 201–210, IEEE Computer Society, 2000.

[Kosc06] Rainer Koschke, Gerardo Canfora and Jörg Czeranski, "Revisiting the approach to component recovery", *Science of Computer Programming*, vol. 60, no. 2, pp. 171 – 188, 2006. Special Issue on Software Analysis, Evolution and, Reengineering.

[Kuhn55] Harold W. Kuhn, "The Hungarian Method for the Assignment Problem", *Naval Research Logistics Quarterly*, vol. 2, no. 1–2, pp. 83–97, March 1955.

[Leth02] Timothy T. Lethbridge and Francisco Herrera, "Assessing the Usefulness of the TKSee Software Exploration Tool", in Hakan Erdogmus and Oryal Tanir, editors, *Advances in Software Engineering*, chapter Metrics, pp. 73–93, Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[Lung98] Chung-Horng Lung, "Software Architecture Recovery and Restructuring Through Clustering Techniques", in *Proceedings of the Third International Workshop on Software Architecture*, ISAW '98, pp. 101–104, New York, NY, USA, 1998, ACM.

[Lung04] Chung-Horng Lung, Marzia Zaman and Amit Nandi, "Applications of Clustering Techniques to Software Partitioning, Recovery and Restructuring", *J. Syst. Softw.*, vol. 73, no. 2, pp. 227–244, October 2004.

[Lung06] Chung-Horng Lung, Xia Xu, Marzia Zaman and Anand Srinivasan, "Program restructuring using clustering techniques", *Journal of Systems and Software*, vol. 79, no. 9, pp. 1261 – 1279, 2006. Selected papers from the fourth Source Code Analysis and Manipulation (SCAM 2004) Workshop Fourth Source Code Analysis and Manipulation Workshop.

[Manc99] S. Mancoridis, B. S. Mitchell, Y. Chen and E. R. Gansner, "Bunch: A clustering tool for the recovery and maintenance of software system structures", in *In Proceedings; IEEE International Conference on Software Maintenance*, p. pages, IEEE Computer Society Press, 1999.

[Maqb07] Onaiza Maqbool and Haroon Babri, "Hierarchical Clustering for Software Architecture Recovery", *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 759–780, November 2007.

[McQu06] Jacqueline A. McQuillan and James F. Power, "Experiences of using the Dagstuhl Middle Metamodel for defining software metrics", in *Proceedings of the 4th International Conference on Principles and Practices of Programming in Java*, 2006.

[Mitc01] Brian S. Mitchell and Spiros Mancoridis, "CRAFT: A Framework for Evaluating Software Clustering Results in the Absence of Benchmark Decompositions", in *WCRE*, pp. 93–102, 2001.

[Mois03] Daniel L. Moise and Kenny Wong, "An Industrial Experience in Reverse Engineering", in *Proceedings of the 10th Working Conference on Reverse Engineering*, WCRE '03, pp. 275–, Washington, DC, USA, 2003, IEEE Computer Society.

[Mull88] H. A. Müller and K. Klashinsky, "Rigi-A System for Programming-in-the-large", in *Proceedings of the 10th International Conference on Software Engineering*, ICSE '88, pp. 80–86, Los Alamitos, CA, USA, 1988, IEEE Computer Society Press.

[Mül93] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley and James S. Uhl, "A Reverse Engineering Approach To Subsystem Structure Identification", 1993.

[Objea] Inc. Object Management Group, "MOF to IDL Mapping, Version 2", http://www.omg.org/spec/MOF2I/.

[Objeb] Inc. Object Management Group, "UML Infrustructure Specification", http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF/.

[Obje07] Inc. Object Management Group, "MOF 2.0/XMI Mapping Vesrion 2.1.1", http://www.omg.org/spec/XMI/2.1.1/, 2007.

[Obje13] Inc. Object Management Group, "MOF Core Specification Vesrion 2.4.1", http://www.omg.org/spec/MOF/2.4.1/, 2013.

[OMG] "Object Management Group, Inc.", www.omg.org/mda.

[Proc02] Java Community Process, "Java Metadata Interface (JMI) Specification, JSR 040, Version 1.0", http://www.omg.org/spec/MOF2I/, 2002.

[Sc01] Tim Littlefair B. Sc, "AN INVESTIGATION INTO THE USE OF SOFTWARE CODE METRICS IN THE INDUSTRIAL SOFTWARE DEVELOPMENT ENVIRONMENT", 2001.

[Selo03] Petri Selonen, "Set Operations for Unified Modeling Language,", in *Proceedings of the Eight Symposium on Programming Languages and Tools, SPLST'2003*, pp. 70–81, 2003.

[SN] "The Source Navigator IDE Homepage", http://sourcenav.sourceforge.net/.

[Tele02] Alexandru Telea, Alessandro Maccari and Claudio Riva, *An Open Visualization Toolkit for Reverse Architecting*, pp. 3–10, University of Groningen, Johann Bernoulli Institute for Mathematics and Computer Science, 2002.

[Tzer97] V. Tzerpos and R. C. Holt, "The orphan adoption problem in architecture maintenance", in *Working Conference on Reverse Engineering (WCRE 1997)*, p. 76, Amsterdam, The Netherlands, Oktober 1997.

[Tzer99] Vassilios Tzerpos and Richard C. Holt, "MoJo: A Distance Metric for Software Clusterings", in *Sixth Working Conference on Reverse Engineering, WCRE '99, Atlanta, Georgia, USA, October 6-8, 1999*, p. 187, 1999.

[Tzer00] Vassilios Tzerpos and R. C. Holt, "ACDC : An Algorithm for Comprehension-Driven Clustering", in *In Proceedings of the Seventh Working Conference on Reverse Engineering*, pp. 258–267, IEEE, 2000.

[Wils86] Robin J Wilson, *Introduction to Graph Theory*, John Wiley & Sons, Inc., New York, NY, USA, 1986.

[Wu05a] Jingwei Wu, Ahmed E. Hassan and Richard C. Holt, "Comparison of Clustering Algorithms in the Context of Software Evolution.", in *ICSM*, pp. 525–535, IEEE Computer Society, 2005.

[Wu05b] Jingwei Wu, Ahmed E. Hassan and Richard C. Holt, "Comparison of Clustering Algorithms in the Context of Software Evolution", in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pp. 525–535, Washington, DC, USA, 2005, IEEE Computer Society.

[Xing05] Zhenchang Xing and Eleni Stroulia, "UMLDiff: An Algorithm for Object-oriented Design Differencing", *Proc. 20th International Conference on Automated Software Engineering*, pp. 54–65, 2005.

[Zhao10] Xulin Zhao and Ying Zou, "A Business Process Driven Approach for Generating Software Architecture", in *Proceedings of the 10th International Conference on Quality Software, QSIC 2010, Zhangjiajie, China, 14-15 July 2010*, pp. 180–189, 2010.

[Zhon04] Shi Zhong, Taghi M. Khoshgoftaar and Naeem Seliya, "Analyzing Software Measurement Data with Clustering Techniques", *IEEE Intelligent Systems*, vol. 19, no. 2, pp. 20–27, March 2004.