Empir Software Eng (2017) 22:1717–1762
DOI 10.1007/s10664-016-9459-z

Case study on which relations to use for clustering-based software architecture recovery

Ioanna Stavropoulou[1] · Marios Grigoriou[1] · Kostas Kontogiannis[2]

Published online: 27 January 2017

Abstract Clustering-based software architecture recovery is an area that has received significant attention in the software engineering community over the years. Its key concept is the compilation and clustering of a system-wide graph that consists of source code entities as nodes, and source code relations as edges. However, the related research has mostly focused on investigating different clustering methods and techniques, and consequently there is limited work on addressing the question of what is a minimal set of relations that can be easily extracted from the system's source code, and yet can be accurately used for extracting its architecture. In this paper, we report on results obtained from an architecture recovery case study we have conducted, by considering all possible combinations which can be generated from thirteen commonly used source code relations. We have examined the similarity of the extracted architectures obtained by using each different relation combination for different systems, against the corresponding architecture which is obtained by applying all thirteen relations and whch we consider as the ground truth architecture. For this purpose, we have also examined whether the use of all these thirteen relations is indeed adequate to yield a ground truth architecture, by applying this architecture extraction process on five large sofware systems for which their ground truth architecture has been independently established. The overall results of our study indicate that there is small set of relations for

Communicated by: Paolo Tonella

✉ Kostas Kontogiannis
kostas@csd.uwo.ca

Ioanna Stavropoulou
ioanna@cs.toronto.edu

Marios Grigoriou
msgrig@softlab.ntua.gr

[1] Department of Electrical and Computer Engineering, National Technical University of Athens, Athens 15780, Greece

[2] Department of Computer Science, Western University, London ON N6A 5B7, Canada

procedural systems, and another similar set for object oriented systems, that can be easily extracted from the source code and yet used to yield an architecture that is close to the ground truth architecture.

Keywords Reverse engineering · Architecture recovery · Clustering · Source code relations · Case study

1 Introduction

Software architecture relates to descriptions that depict high level system design decisions. In a nutshell, software architecture is concerned with the gross organization and global control structure of a system, and aims to bridge the gap between the requirements specifications and the implementation of the system[1] (Bass et al. 2013).

However, as a software system is maintained, its source code evolves, and so does its architecture. In this context, a major challenge software engineers have to deal with, is the problem of ensuring that a software system's architecture is kept up to date with its implementation, as the system evolves. In software engineering, the process of extracting the *as-is* current concrete architecture of an evolving system, by analyzing its source code, is known as software architecture recovery. The motivation for software architecture recovery includes the understanding of the system's structure, the ability to compare the evolved system's architecture, against the initially specified one, the discovery of possible violations of invariants and architectural constraints due to prolonged maintenance, and the design of migration plans so that the system can be ported, or integrated to new platforms, or operating environments.

Software architecture recovery has been a focal point in the software reverse engineering community, and there is a significant volume of related research literature in the subject (Maqbool and Babri 2007; Corazza et al. 2011; Garcia et al. 2011). The approaches proposed in the research literature for software architecture recovery are generally classified as clustering-based (Lung 1998a), domain-based (DeBaud et al. 1994), and structure-based (Koschke et al. 2006).

Even though there is significant volume of work published in clustering-based architecture recovery, there is still limited work on identifying relation combinations that are both easy to extract (i.e. do require a full parser and a linker), and at the same time are the most important ones to use for obtaining in a tractable way an accurate snapshot of the system's architecture. The accuracy of the automatically extracted architecture is defined by its similarity to the actual architecture of the system being analyzed.

1.1 Architecture Recovery Scoping

In the related literature we can identify several definitions for software architecture, most of which consider software architecture as a collection of components and connectors (Bass et al. 2012). However, as suggested by (Kruchten 1995) software architecture can be approached from different viewpoints. More specifically, Kruchten proposed the 4+1 is a

[1] A list of software architecture definitions can be found on http://www.sei.cmu.edu/architecture/start/glossary/classicdefs.cfm.

architectrure model which allows for the system to be represented by different concurrent views that depict and denote the concerns of different stakeholders. The four main views are defined as the logical, development, process and physical view. These views are bound by a fifth view that corresponds to a set of use cases that can be used to illustrate, explain, and validate the overall system architecture.

As it is expected, the recovery of the different software architecture views requires the use of different extraction techniques. For example, the extraction of the logical view of the system's architecture depends on the analysis of requirements models, use cases, and activity models. Similarly, the extraction of the physical and deployment views require the analysis of dynamic information such as logs, network traffic, as well as configuration and installation scripts.

In this paper, we are concerned with the extraction and recovery of the development view of a software system. The development view is a model that represents system components as collections of source code artifacts (e.g. files, functions, types), and system connectors as relations between these system components (e.g. calls, uses, sets). In this context, clustering has been proposed as a suitable method for the recovery of a system's development architectural view.

In a nutshell, the idea of software architecture recovery through clustering is based on a) the analysis of the system's source code; b) the extraction of relations between the system's entities; c) the compilation of a system graph where nodes are source code entities and edges are relations between these entities, and; d) the partitioning through clustering of the system graph into disjoint sets of source code entities, that are considered to be the components of the extracted architecture. For this subject, the reverse engineering community has focused on devising clustering criteria and distances to facilitate cluster formation, experimented with different clustering algorithms, considered the incorporaton of domain knowledge, reflection, and pattern matching as well as, the use of dynamic system information obtained from system logs.

1.2 Motivation and Rationale of the Study

As discussed above, software architecture recovery is an important step in order to perform a number of reverse engineering and re-engineering tasks. These tasks include the analysis of the architectural drift of a system due to prolonged maintenance, the identification of possible architecture violations such as unspecified or undesired interfaces or data exchanges between components, the identification of possible refactoring transformations, and the exposure of legacy components as services over the Internet. Even though software architecture recovery is not a task whis is performed at real time, and usually software engineers have ample time in their disposal to recover the architecture of a legacy system, there are two major stumbling blocks that hinder this effort.

The first stumbling block is the system's size and complexity. Clustering-based software architecture recovery relies on the compilation of an intermediate model of various tuples that represent relations between source code artifacts. This intermediate model is populated and constructed by analyzing the system's abstract syntax tree, or some other representation of the source code. Obtaining, storing and, analyzing an abstract syntax tree or another low level representation of the source code, is not always an easy task. Parsers and linkers are not always readily available, robust, or tractable when applied on large software systems. Among other works in the literature, Akers et al. (2005); Boughanmi (2010); Fleck et al. (2016) discuss the challenges of parsing and analysing large software systems, especially

when these systems are written in different languages, obsolete languages, or involve complex scripting. Storing and analyzing in-memory such large models, may require specialized hardware configurations and computation resources. Experience has shown that for very large systems obtaining a fully linked abstract syntax tree, or loading it in-memory for analysis is not always possible. Software engineering practice has shown that the size of the Abstract Syntaxt Tree compared to the source code size can range from one order of magnitude for small systems, to three orders of magnitude for larger systems (Zou and Kontogiannis 2001; Overbey and Johnson 2008).

The second stumbling block is the system's age and programming languages used. The availability of parsers and linkers for obsolete languages is very limited. In many cases, parsers and linkers have to be tweaked or adjusted, a process that requires significant expertise, it is time consuming, and incolves many trial-and-error steps. For some languages there are not even published EBNF specifications of the programming language grammar itself (e.g. MicroFocus, Adabas, Natural, or variants of PL). Constructing a parser and a linker for a re-engineering project is not an easy or economical task and involves significant risks for the viability of the whole project. Reverse engineering or re-engineering projects are best to be initiated when the availability of the appropriate tools and personnel have been secured (Ducasse and Tichelaar 2003; Tilley et al. 1999).

In this context, the question that arises is *whether we can perform accurate enough architecture recovery of large software systems while minimizing the data sources needed, saving resources and speeding up the process by using less input data that are more easily obtainable from the system's source code*. The answer to this question can provide a valuable tool to software engineers and practitioners, in order to proceed with architecture recovery by depending only on a small set of easily extracted relations that require simple scanners and matchers as opposed to parsers and linkers which are difficult to build, modify, or acquire. Furthermore, the whole process of extracting and analyzing intermediate relation models can become tractable for very large software systems, as the size of in-memory models that have to be analyzed becomes smaller due to smaller number of relations that have to be considered.

1.3 Paper Organization

The paper is organized as follows. Section 2 defines the problem forrmally and outlines the method used. Section 3 provides an overview of different techniques for architecture recovery through clustering. Section 4, discusses the overall evaluation process and the tools used. Section 5 presents the relations that have been considered. Section 6 presents the ACME-based architecture variant metamodel we have used to denote the extracted system architecture, and presents in detail the architecture distance score that has been utilized to assess the dissimilarity of the extracted architecture, from the reference architecture considered as the ground truth architecture. Section 7 discusses the evaluation of the ground truth architectures. Section 8 presents and comments on the results obtained, while Section 9 concludes the paper and provides pointers for future research.

2 Definition of the Study

In this Section, we define the objectives, research questions, method and, evaluation process of our study.

Objective The main objective of the study is to identify a minimal collection of source code relations that can be easily extracted from the source code of the system and at the same tine can be used to recover, within an acceptable level of accuracy, the architecture of a software system. The acceptable level of accuracy is defined and computed by a distance score between the extracted architecture and the ground truth architecture of a system.

Input The case study platform considers three different types of input. The first input is a simplified architecture metamodel denoted in Eclipse Modeling Framework (EMF). The simplified architecture metamodel (M1 level) is based on ACME, an Architecture Description Language developed at CMU. Using the result obtained by the clustering algorithm, this M1 level metamodel is instantiated to yield a M0 level model, that essentially denotes the extracted architecture of the system being analyzed. This architecture metamodel has been drafted for our case study, as the use of other more elaborate and complete metamodels such as the Dagstuhl Middle Metamodel (Lethbridge et al. 2004) would add unecessary modeling complexity given our specific case study goals and objectives.

The second input is a set of different relation combinations which will be used to perform clustering in order to extract the architecture of the system being analyzed.

The third input is a differencer that computes a distance score between the extracted architecture represented by a M0 level model for a given relation combination, and the ground truth architecture which is also represented as an M0 level model.

Output The output of the system is a distance score between the extracted M0 level architecture model that has been extracted using a specific combination of relations, and the M0 level architecture model that is considered as the ground truth architecture of the system. The distance score for each relation combination and for all systems are analyzed to answer our specific research questions.

Research Questions In this study we aim to answer four main research questions by examining a number of procedural and object oriented systems. A fifth question to be answered is whether we observe any difference between the relations required to extract the architecture of a procedural system and the relations required to extract the architecture of an object oriented system.

The first research question is whether *there are any relations that are the most important and should be always used when available for architecture recovery*. The answer to this research question will identify collections of relations that are the most effective on recovering a system's architecture. The second research question is whether *there are any relations that do not offer any apparent value for architecture recovery*. The answer to this question will help on the identification of relations that do not offer any value (positive or negative), and therefore can be safely omitted. In this respect, the software maintenance team does not need to spend time and effort extracting these relations from the system's source code. The third research question is whether *there there any relations that should not be used, or not be used on their own*. The answer to this question will help on the identification of relations that when are used, or when are used on their own, will have as an effect the introduction of noise in the extracted architecture. Finally, the fourth research question is to evaluate *how many relations do we need for accurate architecture recovery*. The answer to this question will identify a minimum collection or collections of relations that are best to use for architecture recovery. In this respect, we are also interested to identify those relations that are easy to extract from the system's source code. The ease of extraction is determined by the

method that is required to extract a given relation. If a relation can be extracted by a scanner or a simple pattern matcher (e.g. grep), then it is considered an easy to extract relation. If on the other hand, a relation can be extracted only by using an AST and requiring the use of full parser and a linker, then the relation is considered hard to extract.

Method The method used in this study comprises two main steps.

The first step relates to the extraction of different variants of the architecture of a subject system, one variant for each possible relation combination. For this study we have considered thirteen commonly used source code relations, which produced 8192 different combinations. First, all combinations containing one relation are considered, then those combinations containing two relations, then those involving three relations, and so on. To commence clustering and extract an architecture for a given relation combination, we have kept in the RSF file only the relations which participate in this particular combination.

The second step relates to the computation of a distance score between the extracted architecture and the architecture we consider as ground truth. The extracted architecture and the ground truth architecture are represented as M0 level models that are compliant with our ACME-based M1 level metamodel. A more detailed description of the architecture extraction and differencing process is presented in Section 4 and in Section 6. The results obtained by all possible relation combinations are analyzed for their statistical significance as to whether the distance scores and the relation combinations are independent, using Pearsons Chi-Square test. In this respect, the Null hypothesis H0 is that *distance measures are independent of the relation combinations used*. Consequently, the Alternative hypothesis H1 is that *distance measures are dependent on the relation combinations used*. This test wil allow us to conclude whether there is a strong relation between the obtained distance scores and the relations used, or these scores are obtained by pure chance.

Ground Truth Architecture Validation A key aspect of our study is the computation of a difference score between an extracted architecture using a given relation combination, and the architecture we consider as the ground truth architecture. The premise we take in this paper is that the application of the thirteen commonly used relations is adequate for extracting the ground truth architecture. One way of showing this is to manually examine, for each of the fifteen systems, the architectures that have been obtained by using all thirteen relations, and verify that these match the corresponding ground truth architectures. The analysis can be based on comparing directory structure, naming conventions, published manuals when available, and software engineering design expertise. Even though this approach provides good evidence for deciding whether or not to consider an architecture as a ground truth architecture, at the same time may still be prone to our subjective interpretations and errors, as not all information can be independently available for establishing the ground truth. Instead, we opted for an approach that aims to extract the architecture of large third party systems using our thirteen relations, and consequently compare whether the extracted architecture using all thirteen relations matches at an acceptable level the published and authoritative ground truth architecture of these systems. In this respect, we have considered five systems, the ground truth architectures of which have been reported by Lutellier et.al. (2015). A detailed discussion on the evaluation of ground truth architectures is presented in Section 7.

3 Related Work

3.1 Architecture Recovery Using Clustering

Software architecture recovery has been a major focus of the reverse engineering research community for more than two decades (Mancoridis and Holt 1996; Anquetil and Lethbridge 1998). In this respect, several automated tools have been proposed, based on areas such as artificial intelligence, programming languages and, graph theory.

Among the most investigated techniques for the recovery of a system's development architectural view are the ones that are based on source code static analysis and clustering.

The software engineering research literature on architecture recovery using clustering is too rich to exhaustively refer to here, but notable approaches include the work presented by Bauer and Trifu (2004), Canfora et al. (2000), Chiricota et al. (2003), van Deursen and Kuipers (1999), Maqbool and Babri (2004), Sartipi and Kontogiannis (2003), Lung (1998b), Mahdavi et al. (2003), Muller et al. (1992). These techniques group entities into clusters, where each cluster represents a component of the software system's architecture. One of the first approaches proposed was by Mancoridis and Holt (1996), which utilized graphs created during the design specification phase of a software system.

The Algorithm for Comprehension-Driven Clustering (ACDC) by Tzerpos and Holt (2000a) was proposed as a technique that provides a decomposition of the software system according to patterns that are commonly found on manually created decompositions of software systems. Similarly, a dependency-based software clustering algorithm is proposed by Kobayashi et al. (2012), and an algorithm that achieves a decomposition of a software system using artificial intelligence algorithms such as hill climbing, or genetic algorithms, is proposed by Mancoridis et al. (1999).

Another approach in cluster-based architectural extraction is presented by Maqbool and Babri (2007), and by Andritsos and Tzerpos (2005). These techniques allow for the software engineer to obtain an architecture view at different levels of abstraction. More specifically, the scaLable InforMation BOttleneck (LIMBO) approach by Andritsos in Andritsos and Tzerpos (2005) is based on the minimization of information loss during the clustering process. Furthermore, (Corazza et al. 2011) propose the use of lexical information for software architecture recovery. A detailed evaluation of the different clustering techniques for architecture recovery is presented by Tzerpos and Holt (2000b), while the impact call graphs have on the clustering-based architecture recovery is presented by Rayside et al. (2000).

3.2 Architecture Recovery Using Other Methods

In addition to software architecture recovery using clustering, a number of other methods have also been proposed in the related software engineering literature. These can be generally classified as methods that are based on dynamic analysis, analysis of evolution changes, reflection models, pattern matching analysis of templates, analysis of product families, and methods based on visualization and interactive user involvement.

More specifically, in the area of dynamic analysis (Vasconcelos and Werner 2004) propose a system that collects system traces for specific use cases. These system traces allow for the identification of interaction patterns and ultimately the extraction of system components. In a simlar approach, (Bojic and Velasevic 2000) utilize traces of test cases that cover relevant use cases. The analysis of these traces using formal concept analysis allows

for the selection of parts of the system that implement similar functionality. Patel et.al. (2009) propose a two-phase software architecture recovery technique that combines static and dynamic analysis. In the first phase, an initial architecture is extracted using static information obtained from the source code. In the second phase, the extracted architecture is refined using system traces obtained from the execution of specific use cases. (Jerding and Rugaber 2000) proposes a tool that utilizes both static and dynamic analysis to extract and visualize the system's architecture, while (Garcia et al. 2011) presents a technique that uses a representation of the system's concerns, information retrieval and machine learning to recover the software architecture.

Another method used for architecture recovery is based on the analysis of evolution changes and the computation of change coupling among system components. In the work presented by Fischer et al. (2003), system features are tracked as the system evolves, and are visualized in order to identify hidden dependencies between system components.

Reflection models have been proposed by Murphy and constitute an important method for architecture analysis. In particular, (Murphy et al. 1995) presents a framework that allows for the computation of a reflection model that illustrates where an engineer's high-level achitectural model matches or corresponds with the system's implementation. As an extension of reflection models, pattern matching techniques have also used for software architecture recovery. Sartipi and Kontogiannis (2001) presents a pattern language that generates a pattern graph. A matching process then aims to optimaly instantiate the pattern graph with information obtained by the source code of the system being analyzed. Another approach in a similar direction is presented by Bowman and Holt (1998) who proposes that the ownership structure of a software system can provide valuable information about its architecture. Mendonça and Kramer (2001) discusses X-ray, a tool that combines pattern matching, structural reachability analysis and component module classification, in order to recover the architecture of distributed systems.

For systems that are part of product families and product lines, the analysis of common patterns and styles in the product family constitutes an effective method for architecture recovery. Pinzger et al. (2004) present a framework which allows the recovery of a reference architecture from related prior systems in a product family.

Lungu et al. (2014) present an architecture recovery technique that is based on interactive user involvement and the visualization of software artifacts. Tzerpos and Holt (1996) propose a hybrid approach that uses the information acquired by system developers in order to validate and fine tune the information that is automatically extracted from the source code. Additionally, Garcia et al. (2013) proposes the involvement of one or more of the system's architects or engineers in order to enhance the results obtained by the automated tools.

3.3 Architectural Models

In order for Software Architecture to be modeled, a number of Architecture Description Languages (ADLs) have been proposed. Most ADLs employ both a graphical and a textual syntax. Some of the most well known ADLs are:

Architecture Analysis and Design Language (AADL) is designed for the specification, analysis, automated integration and code generation of real-time performance-critical distributed computer systems (Feiler 2014).

ACME is a simple, generic software ADL developed at CMU. It can be used as a common interchange format for architecture design tools, and as a foundation for developing new architectural design and analysis tools (Garlan et al. 1997).

Alloy is a language developed at MIT for describing structures as a collection of constraints (Jackson 2012).

C2 is a language developed at UCI to describe C2 styled architectures. C2 is a component and message-based architectural style for constructing flexible and extensible software systems (Medvidovic 1995).

Wright is an ADL developed at CMU, and formalizes a software architecture in terms of concepts such as components, connectors, roles, and ports (Allen 1997).

3.4 Evaluation of Architectural Recovery Techniques

The architecture recovery techniques we have presented above have been evaluated for their accuracy on several studies by Wu et al. (2005b); (2005a); Garcia et al. (2013), and by Andritsos and Tzerpos (2005). Similarly, a very interesting process-oriented taxonomy of software architecture reconstruction techniques is presented by Ducasse in (Ducasse and Pollet 2009).

The two most recent studies that evaluate clustering based techniques examine the problem from two different angles. Garcia et al. (2013) evaluate six different architecture extraction techniques on various software systems. More specifically, six techniques, ACDC, ARC, Bunch, LIMBO, WCA and ZBR were evaluated on open source systems such as Arch-Studio, Bash, Hadoop, Linux, Mozilla and OODT, with ACDC and ARC showing the most promising results. On the other hand, (Lutellier et al. 2015) evaluate the impact different dependencies, namely *include* dependencies and *structural* dependencies, have on the architectural recovery techniques mentioned above. The outcome is that structural dependencies provide better outcomes than ”include” dependencies, and still ACDC and ARC produced the best results, compared to the other techniques. Other studies, such as the one presented by Tzerpos and Holt (2000b), evaluate different aspects of software architecture recovery techniques such as stability, while (Wu et al. 2005b) evaluates architecture recovery techniques based on authoritativeness and extremity of cluster distribution.

Although extensive work has been conducted on devising and evaluating architecture extraction techniques, there is limited work concerned with which relations are the most appropriate to use (Lutellier et al. 2015). More specifically, (Lutellier et al. 2015) focus on the comparison on ”include” dependencies to symbol dependencies, without taking into account how different relations, or relation combinations, may affect architecture recovery, which is the focal point of our paper.

4 Outline of the Analysis Process

In this section, we outline the evaluation framework process. As depicted in Fig. 1, the process consists of four phases.

The first phase is the extraction phase. During the extraction phase, the source code of the system is parsed using the Fetch tool (Fact Extraction Tool Chain). The Fetch tool chain has

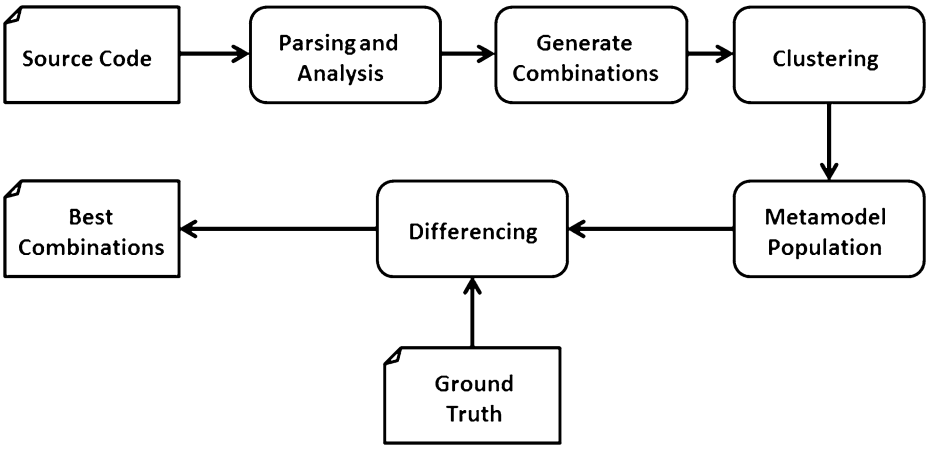

Fig. 1 The chain of process steps comprising the implemented framework

been developed by the Re-engineering group at the University of Antwerp (Bois et al. 2007), and we used it in order to produce CDIF, an intermediate model of the source code being parsed (Imber 1991). A CDIF-to-RSF transformer is then used to link and resolve CDIF entries, and produce Rigi Standard Form (RSF) tuples. Figure 2 depicts how Fetch combines different open source tools to generate CDIF specifications. We have opted to use the Fetch tool chain because of its robust parser, and analyzer. Furthermore, the tool chain is composed of open source components, we could extend where required to obtain better static analysis results, as we actually did for analyzing *typedef* statements in C++ systems. We have not been able to find a Source Navigator to RSF transformer, in order to bypass CDIF and speed up the extraction process, so we have kept the original configuration of Fetch tool chain. Recent approaches that have successfully used though this tool chain, include the work by Adams et al. (2009) and (Van Rompaey and Demeyer 2009; Van Rompaey et al. 2009).

The second phase is the clustering phase. During the clustering phase, a subset of all the source code tuples that correspond to a combination (i.e. subset) of all available relations is

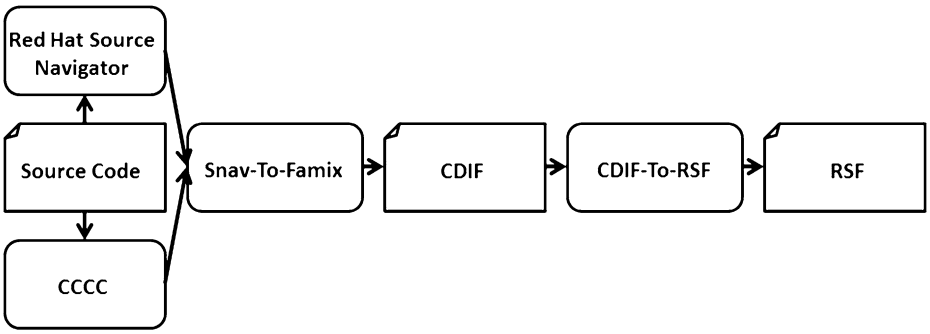

Fig. 2 Fact Extraction Tool Chain

selected. Consequently, only these tuples are fed to the ACDC software architecture extraction clustering tool in order to produce a set of clusters which are composed of source code entities such as functions, variables and, types. Each such cluster, corresponds to an *extracted* architectural component.

The third phase of the process is the M0 level architecture model creation phase. In this phase, our ACME-based M1 level metamodel is instantiated so that entities, components, connectors and bindings are created utilizing the clustering results obtained in the previous phase. The ACME-based metamodel is discussed in more detail in Section 6 below.

The fourth phase of the process is the similarity evaluation phase. In this phase, an architecture model distance score between the instantiated architecture model extracted using a combination (i.e. a subset) of the available relations, and the instantiated architecture model that has been extracted using *all* available relations (and being considered as the reference standard), is computed. The normalized distance score is a real number that ranges from 0 (i.e. two architecture models are completely similar) to 1 (i.e. two architecture models are completely dissimilar).

Each relation combination is created in a specific order. First, all combinations containing one relation are considered, then those containing two relations, then those involving three relations, and so on. For our system we considered thirteen relations, which produced 8192 different combinations. The thirteen different relations we have considered are depicted in Table 1. We will elaborate on these relations on Section 5.2 below.

For our work, we have considered and compared two clustering algorithms, the ACDC algorithm by (Tzerpos and Holt 2000a) the Hill Climbing clustering based algorithm from the Bunch clustering suite by Mancoridis et al. (1999). The choice of the clustering algorithm was made on the criteria of a) clustering time performance and; b) stability.

Both clustering frameworks presented advantages and disadvantages. As (Wu et al. 2005b) have indicated, the ACDC algorithm is more stable than Bunch a tool which incorporates a heuristic clustering process. On the other hand, Bunch produces much more uniform clusters meaning that the sizes of the clusters do not present extremities (not too big, or too small clusters). For small systems, less than 60 KLOC, the performance of the two clustering algorithms is comparable but still significant. However, in bigger systems, 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, for OpenSSL Bunch requires 135 minutes and a program heap size of 6GB. As a result, we have chosen the ACDC algorithm as it was shown to scale better for large systems.

5 Source Code Relation Extraction

5.1 Parsing

Initially, the Source Navigator tool performs lexical analysis and parsing. The SnavToFamix tool creates an AST denoted in CDIF (Imber 1991), which consequently is used to produce an RSF file. We present an example of the use of Fetch for a small C program 1 given in Listing 1.

Once a CDIF file as the one presented in Listing 2 is created, it can be transformed to RSF. The corresponding RSF file is presented in Listing 3. For our case study, we have opted to exclude system standard libraries (e.g. *stdlib.h*), and include only the libraries that are specific to the application being considered.

Table 1 Relations extracted by the Fetch tool

Type	Relation
Containment	– "Module Belongs to Module"
	– "File Belongs to Module"
	– "Class Belongs to File"
	– "Invocable Entity Belongs to File"
	– "Method Belongs to Class"
	– "Attribute Belongs to Class"
Macros	– "Macro Definition"
	– "Macro Use"
Conditions	– "Conditional Compilation"
Location	– "Entity Location"
	– "Entity Belongs to Block"
Declaration	– "Defined In"
	– "Declared In"
Invocations	– "Calls"
Types	– "Type Definition"
	– "Uses Type"
	– "Has Type"
	– "Has Type Definition"
Information	– "Signature"
	– "Visibility"
	– "No of Lines"
Other	– "Inherits From"
	– "Includes"
	– "Accesses" (Variable)
	– "Sets" (Variable)

5.2 Source Code Extracted Relations

The resulting CDIF file encompasses information that can be used to extract a number of different relations. Overall, from CDIF, we can extract 22 different relations and generate RSF tuples for them. However, from these relations we opted to use only 13 of them.

The choice of these relations was based on the concept of omitting relations that offer limited or, overlapping information, for extracting the architecture of a software system. For example, the information of whether a method is abstract or not, or what is its signature, or what is the number of source code lines for *methods* or *structures*, may not be as relevant for the architecture extraction process or a subsumed by other relations, and are consequently omitted. The thirteen relations that we retained for the case study, are depicted in Table 2.

Similarly, Table 3 depicts an assessment of how easy or difficult is to extract each relation from the source code, based on whether it requires the use of scanner, parser or, a parser and a linker.

The extraction of a relation is considered easy if the source code does not need any special tools to extract it. An example is the *Includes* relation, which can be obtained with a simple "grep", on *#include* preprocessor directives. The extraction of a relation is considered of medium difficulty, if a small preprocessing is required, such as for the *Calls* relation,

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

which first requires the identification of which tokens correspond to function names, and then extract the *Calls* relation between them using a scanner. For example there are languages like PL/I, where a procedure call or an array reference look identical. Note that the *Calls* relation denotes the existence of a call statement in the body of a function or a method, and does not deal with the resolution of which method is actually called in the presence of polymorphism. Finally, relations that require the use of a parser, or require information from an annotated AST (Abstract Syntax Tree), are considered hard to extract compared to relations that can be extracted by a scanner or a simple matcher (e.g. grep). For example the *Has Type* and *Uses Type* relations are hard to extract as, in many cases, we have to consider that analysis of *typedef* statements, cases of polymorphism (in object oriented systems), and possible conditional preprocessor directives.

6 Architecture Representation and Differencing

In this section, we present a simplified software architecture metamodel, which we have designed for facilitating our case study. The simplified metamodel is depicted in Fig. 3 and is based on the ACME (Garlan et al. 1997) Architecture Description Language.

The metamodel contains elements which can either be *Components* or *Connectors*. Furthermore, an element, can have several *Properties* and depending on its type, the element contains different *Entities*.

Components are represented by clusters (i.e. sets of source code entities) as these clusters are obtained by the clustering process. Each *Component* has a unique name and is composed of *Entities*. The *Entities* that are attached to a *Component* are organized in a tree structure, based on the relations that connect them. For example, if there is a relation $\langle class, ClassBelongsToFile, file \rangle$, then the *class* entity would be a child of the corresponding *file* entity.

Entities can have *Properties* that characterize them. The type of a class, the file where a method or a function are declared and, the class that a class inherits from, are some of such properties. *Properties* are also children elements of the *Entities* that they are related to.

```
(: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 CDIF for Pascal Triangle

Connectors represent links between *Components*, but they do not provide any information about which specific individual *Entities* they connect. For this reason, each *Connector* is associated with one or more *Bindings*. *Bindings* connect specific *Entities* that belong

```
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 3 RSF for Pascal Triangle

to the *Components* that the *Connector* links. Each *Binding* is characterized by its type (i.e. Call, Access, Include, Set Variable), and the two specific *Entities* that the *Binding* connects to.

By instantiating this metamodel, concrete architecture models are created for each different system and for each different relation combination. This metamodel instantiation procedure is discussed in more detail in Section 6.1 below.

Table 2 Description of relations used

Relation	Description
"Calls"	Denotes that a function or a method calls another function or method.
"Includes"	Denotes that a file includes another file.
"Sets"	Denotes that an attribute in a class, or a global variable, is assigned a value in a function or a method.
"Accesses"	Denotes that the value of an attribute in a class, or of a global variable, is read in a function or a method.
"Class Belongs To File"	Denotes that a class belongs (defined) in a file.
"Inherits From"	Denotes inheritance between classes in object oriented programs.
"Has Type"	Denotes that a function, method, attribute, or global variable is of a specific type.
"Defined In"	Denotes that a function or a method is defined in a specific file. 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 major difference. "Declared In" denotes 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.
"Attribute Belongs To Class"	Denotes that an attribute belongs (defined) in a specific class.
"Method Belongs To Class"	Denotes that a method belongs (defined) in a specific class.
"Accessible Entity Belongs To File"	Denotes that a global variable belongs (defined) in a specific file.
"Uses Type"	Denotes that a method or a function uses (accesses) a specific type (i.e. the type of a class or a struct).

Table 3 Ease of extraction of relations from source code

Relation	Easy	Medium	Hard
"Accesses"		√	
"Accessible Entity Belongs to File"	√		
"Attribute Belongs to Class"		√	
"Class Belongs to File"	√		
"Calls"		√	
"Declared In"		√	
"Defined In"		√	
"Has Type"			√
"Includes"	√		
"Inherits From"	√		
"Method Belongs to Class"	√		
"Sets"		√	
"Uses Type"			√

6.1 Architecture MetaModel Instantiation

Our case study is based on calculating a distance score between *a)* the concrete architectures extracted via clustering using different combinations of the thirteen available relations, and *b)* the concrete architecture we consider as the reference standard and is the one extracted using *all* thirteen available relations. In order to create such concrete instances of the architecture metamodel, for a given subset of relations, we follow a five step process as depicted in Fig. 4.

In the first step, the RSF file is fed to the ACDC clustering process, and clusters c_i made of source code entities sc_j (e.g. files, classes, methods, functions, variables, types) are created. For each such cluster c_i, a *Component* instance C_i, as specified in the architecture metamodel, is created.

In the second step of the process, each source code entity sc_j, in a cluster c_i is used in order to create a corresponding *Entity* type metamodel instance $e_{i,j}$. Furthermore, in this process step, each *Component* C_i created in the first step, is populated with the corresponding $e_{i,j}$ *Entities*.

In the third step of the process, *Connectors* between *Components* are created. A *Connector* $CN_{k,m}$ is created between component C_k and component C_m if there is an *Entity* $e_{k,j}$ in component C_k and an *Entity* $e_{m,w}$ in component C_m, for which there is an RSF tuple $\langle\ sc_j$ a_relation $sc_w\ \rangle$ between the source code entities sc_j, and sc_w, for which the entities $e_{k,j}$ and $e_{m,w}$ correspond to.

In the fourth step of the process, bindings are created between *Entities* $e_{k,j}$ and $e_{m,w}$, in each connector $CN_{k,m}$ and for which entities there is a corresponding RSF tuple $\langle\ sc_j$ a_relation $sc_w\ \rangle$. The Binding becomes a child of its corresponding Connector.

In the fifth step of the process, *Properties* are added to *Entities*. The relations that yield properties are *Uses Type*, *Has Type*, *Inherits From* and *Defined In*. Once this is done for the relations being considered, an instance architecture model is completed and is ready to be compared to the instance architecture model that corresponds to the ground truth reference standard.

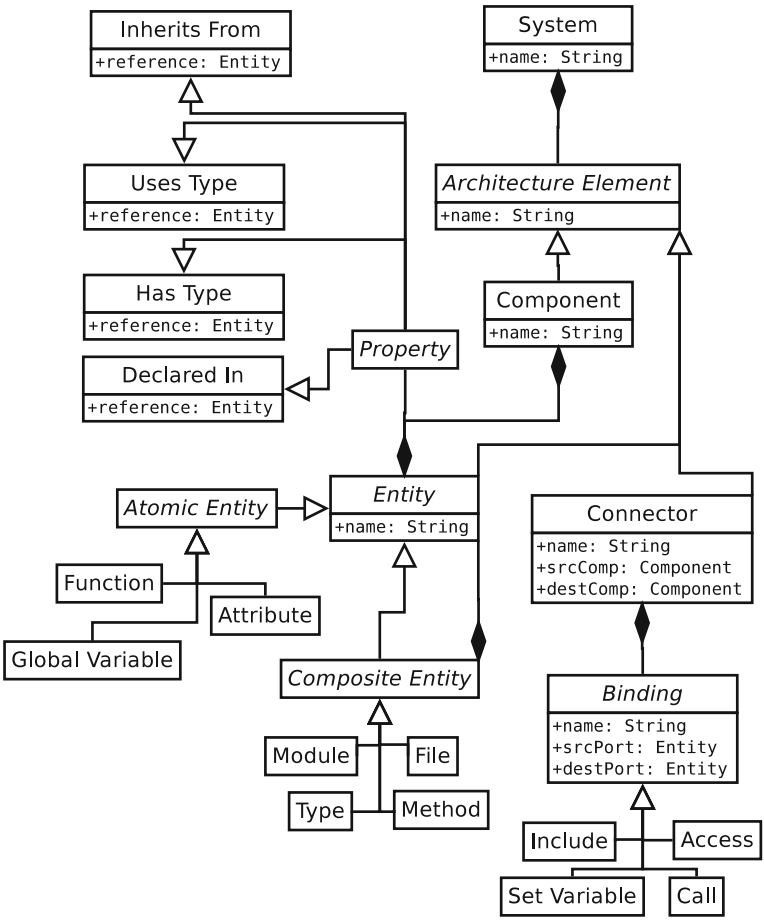

Fig. 3 M1 level architecture metamodel

Figure 5 depicts an example of such an architecture instance. The example architectrure instance is composed of two Components, and one Connector. The first Component *C1*, consists of two files *F1* and *F2*. Similarly, the second Component *C2*, consists of two files *F3* and *F4*. The Connector *CN1* connects the Component *C1* and *C2* and associates with an *Includes* type Binding between Entities *file* Entities *F1* and *F3*, via a ⟨ F1 includes F3 ⟩ relation. Listing 4 illustrates the XML source that corresponds to the instantiated model of the aforementioned example architecture instance.

6.2 Architecture Model Differencing

In this section, we discuss in more detail the similarity measure we have applied in order to compute the structural distance between two software architecture instance models. More specifically, given two instances of software architecture models, we devise a variation of the UMLDiff algorithm presented by Xing and Stroulia (2005), in order to calculate the structural difference of the two model instances.

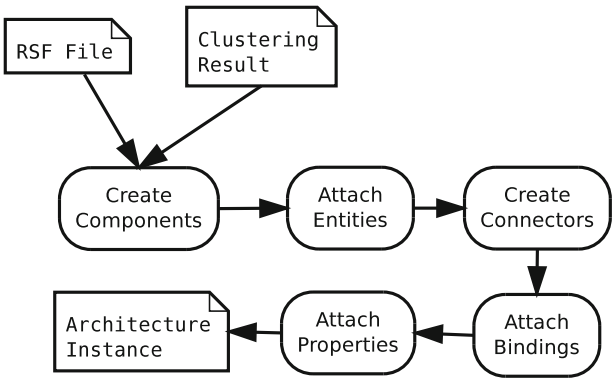

Fig. 4 Steps for Population of Architectural Instances

The algorithm operates by indicating the entities that need to be moved, added, and deleted in order for the first instance to be transformed to the second. The number of insertions, deletions, and substitutions provides the basis for computing a measure of structural difference.

UMLDiff takes as input two XML files denoting two models, and retrieves the structural changes that occurred, as these models evolved from one version to another. UMDiff utilizes a name similarity and a structure similarity component. *Name Similarity* calculates the similarity of the names of the entities by calculating the common adjacent character

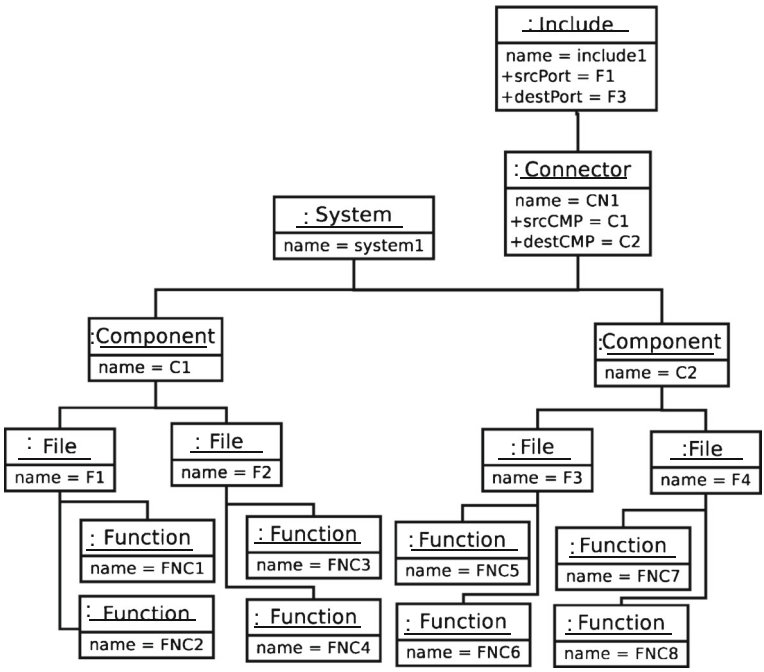

Fig. 5 Example instance architecture 1

```
<?xml version="1.0" encoding="ASCII"?>
<model:System xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:model="http://acme/1.0" name="system1">
<ContainsComponent xsi:type="model:Component" name="C1" level="1" parent="system1">
  <ContainsFile xsi:type="model:File" name="F1" level="2" parent="C1">
    <ContainsFunction xsi:type="model:Function" name="FNC1" level="3" parent="F1"/>
    <ContainsFunction xsi:type="model:Function" name="FNC2" level="3" parent="F1"/>
  </ContainsFile>
  <ContainsFile xsi:type="model:File" name="F2" level="2" parent="C1">
    <ContainsFunction xsi:type="model:Function" name="FNC3" level="3" parent="F2
    "/>
    <ContainsFunction xsi:type="model:Function" name="FNC4" level="3" parent="F2
    "/>
</ContainsFile>
</ContainsComponent>
<ContainsComponent xsi:type="model:Component" name="C2" level="1" parent="system1">
  <ContainsFile xsi:type="model:File" name="F3" level="2" parent="C2">
    <ContainsFunction xsi:type="model:Function" name="FNC5" level="3" parent="F3"/>
    <ContainsFunction xsi:type="model:Function" name="FNC6" level="3" parent="F3"/>
  </ContainsFile>
  <ContainsFile xsi:type="model:File" name="F4" level="2" parent="C2">
    <ContainsFunction xsi:type="model:Function" name="FNC7" level="3" parent="F4"/>
    <ContainsFunction xsi:type="model:Function" name="FNC8" level="3" parent="F4"/>
  </ContainsFile>
</ContainsComponent>
<ContainsConnector xsi:type="model:Connector" name="CN1" level="1" parent="system1">
  <ContainsInclude name="include1" level="2" parent="CN1" srcComp="C1" srcPort="F1"
      destComp="C2" destPort="F3"/>
</ContainsConnector>
</acme.model:System>
```

Listing 4 Component Matching

pairs that are contained in the two compared names. If the similarity score is lower than a threshold provided, the entities are identified as renamed, otherwise are identified as similar. *Structure Similarity* computes the similarity of two entities based on their connection with already matched entities.

Our UMLDiff variant algorithm operates from the root level model element, moving down the model structure on to Components, Composite Entities and finally Atomic Entities. At each level, except the top 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 in Section 6.2.1 below.

Matched entities have the same name and also belong to already matched Components. Moved entities have the same name but they belong to Components that are not matched. Finally deleted entities are the ones that are present on the reference model instance used as the ground truth architecture, but not on the instance under examination. Once all entities have been identified, their properties need also to be matched.

```
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 5 Component Matching

Properties of matched entities are evaluated on whether they are of the same type and on whether the attributes of their properties are also the same. Consequently, these Properties are identified as matched. Otherwise, properties of the ground truth instance are categorized as deleted, and the ones of the instance being examined, 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 considered to be a match. Otherwise, Bindings that have different types, or they refer to different entities, or belong to not matched Connectors are classified as deleted, if they belong to the ground truth instance and, as added if they belong to the instance being examined.

6.2.1 Component Matching

Component matching is handled as a more special case than the Entity, Property, Connector and Binding cases. The reason is that Components are top elements in the model hierarchy and as such, once two Components (one from the reference architecture, and the other from the extracted architecture that is to be evaluated) are identified as match, then this choice affects all other model entities that are linked as children of the two Components, and consequently affects their matching result as well. In a nutshell, the problem is to identify which Component of the reference architecture matches the most with which component of the extracted architecture that is to be evaluated. Once this choice has been made, matching for the dependent model entities can commence.

In this respect, the problem of computing the structural distance between two Components is equivalent to evaluating the structural distance between their two corresponding instance models. This problem can be identified as a maximum weighted bipartite matching (Wilson 1986). The two partite sets are the instance models, one representing a Component of the architecture extracted by a subset of available relations and is under examination, and the other representing a Component of the reference ground truth, that is the architecture extracted by using all available relations. 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. In this respect, we aim to maximize the number of entities that are matched between two Components. This matching problem can be solved using the Hungarian Method (Kuhn 1955). The 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 execution time by decreasing n and actually solving a smaller problem. The outline of the component-level matching algorithm is depicted below.

6.2.2 Entity and Property Matching

The entity-level and property-level matching process commences once the component level matching process completes. Listing 5 outlines the algorithm that is used for identifying whether an entity or a property in one model is classified as matched, moved or, added with respect to the reference architecture model. More specifically, the process at line 2 aims to retrieve an Entity *e2* from the ground truth, that has the same name as *e1*. If such Entity, or Property exists, and both *e1* and *e2* belong to the same logical level of the architecture,

```
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 6 Entity and Property Matching

then we have a *match* (lines 5-6). Otherwise, if they belong to different levels, then this 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 result, once all Entities and Properties have been classified, *setB* contains all the *deleted* Entities and Properties. We have analyzed our results as to whether there are particular relations that contribute to entity *insertion*, *deletion* or *move* operations, while comparing the ground truth architecture and the extracted architecture, but we were not able to observe any significant correlation in order to conclude that one or more relations are the ones which are most responsible for such *insertion*, *deletion* or *move* operations. However, our analysis indicated that Connectors, and not Component entities, are the elements that are mostly inserted or deleted during the matching process.

6.2.3 Connector Matching

After Components, Entities and Properties have been classified as *matched*, *moved* or, *added*, the next step is to match Connectors. Listing 7 depicts the connector-level matching process. The connector-level matching process iterates through all the Connectors of the architecture model under evaluation and attempts to locate a matching Connector in the reference model used as the ground truth architecture. In order for two Connectors, CN_1 and CN_2 to be matched, the components that CN_1 connects need to have already been matched to the Components that CN_2 connects (line 8). The remaining connectors are identified as *added*, if they belong to the examined instance (line 12), and as *deleted* if they are part of the ground truth (line 13), but not part of the model being evaluated.

```
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 (isMatch(c1.components, c2.components))
7  matchedConnectors.add([c1, c2]);
8  conA.remove(c1);
9  conB.remove(c2);
10 added.addAll(conA);
11 deleted.addAll(conB);
```

Listing 7 Connector Matching

6.2.4 Binding Matching

Listing 8 depicts the binding-level matching process. The bindings-level matching proceeds by iterating through all the Bindings of the instance model being evaluated (line 2) and attempting to locate a Binding within the ground truth (lines 3-4) having the same name. If these two Bindings also have matched Connectors as parents, then they are identified as *matched* Bindings (lines 5-7), otherwise they are identified as *moved* (lines 9-10). If there is no such Binding in the reference model, then the Binding is identified as a *deleted* Binding from the reference ground truth architecture. Whenever a move or a match is identified, the Binding is removed from the set of Bindings of the ground truth, and as a result all remaining Bindings in that set are identified as *added* Bindings.

6.2.5 Differencing Score

The output of the algorithms presented in the previous sections (Listings 5 - 8), aim to identify all the elements that match. Once the number of elements that match is computed, the overall structural distance between two instance architecture models for systems S_i and S_j can be computed by applying Eq. 1:

$$diff_{i,j} = \frac{Total\_Number\_of\_Elements_{i,j} - Number\_of\_Matched\_Elements_{i,j}}{Total\_Number\_of\_Elements_{i,j}} \quad (1)$$

Even though other distance scoring functions could be considered (e.g. assigning weights to different relations), we have opted for this simple, yet effective, function, that is impartial to the type and perceived importance of the types of the elements compared. This distance measure aims to fulfill certain criteria. The first criterion deals with stability, meaning that small structural model changes should reflect proportionaly small score changes. The second criterion is that once scores are normalized, same models should yield a difference score equal to 0, while completely dissimilar models should yield a distance score equal to 1. Score normalization for a given system S_i for relation combination R, is achieved by applying Eq. 2.

$$normalized\_diff_{R,i} = \frac{|diff_{R,i} - min(diff_{all\_R,i})|}{max(diff_{all\_R,i}) - min(diff_{all\_R,i})} \quad (2)$$

```
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 8 Binding Matching

where $min(diff_{all\_R,i})$ is the smallest distance score obtained for system S_i accross all possible relation combinations for system S_i, and $max(diff_{all\_R,i})$ is the corresponding highest distance score.

6.2.6 *Differencing Score Example*

In this section, we present a simple example that depicts how the distance score is calculated by following the process discussed in the previous sections.

For illustration purposes, let us consider two instance architectures depicted in Figs. 6 and 7 respectively. Let us further assume that the architecture instance depicted in Fig. 6 is considered the ground truth reference standard. The distance score is then calculated as follows:

Step 1 *Match Components*
Each Component of the architecture under evaluation is compared with the Components from the ground truth. In this respect, C1 (in Fig. 6) is matched with C1 (in Fig. 7) and C2 (6) is matched with C2 (7). The matching between these sets is based on the maximum number of same name elements appearing in each Component (i.e. corresponding cluster). The remaining Component C3 in the architecture depicted in Fig. 7, is considered an added Component.

Step 2 *Match Entities and Properties*
As we discussed previously, Entities and Properties are matched based on their name and their parent Entity, since names are unique across the system. As a result,

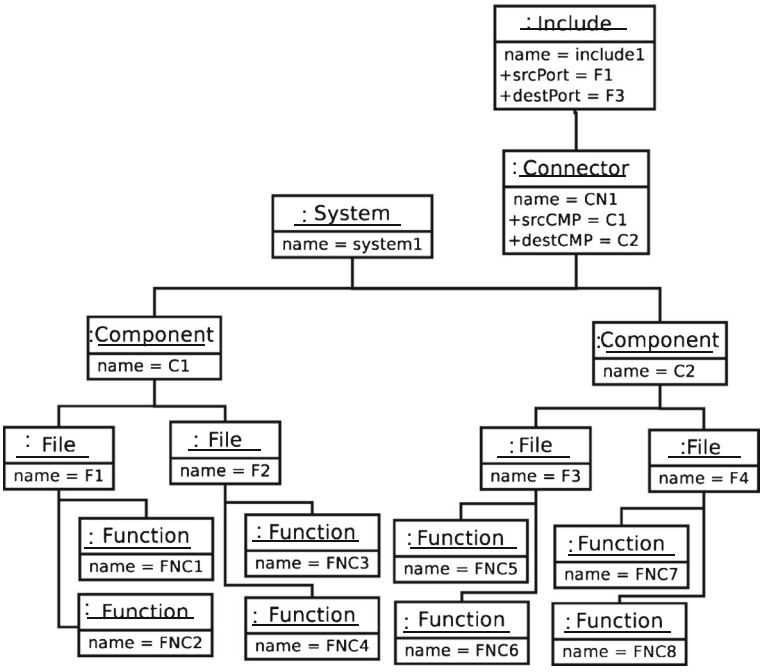

Fig. 6 Example Architecture 1

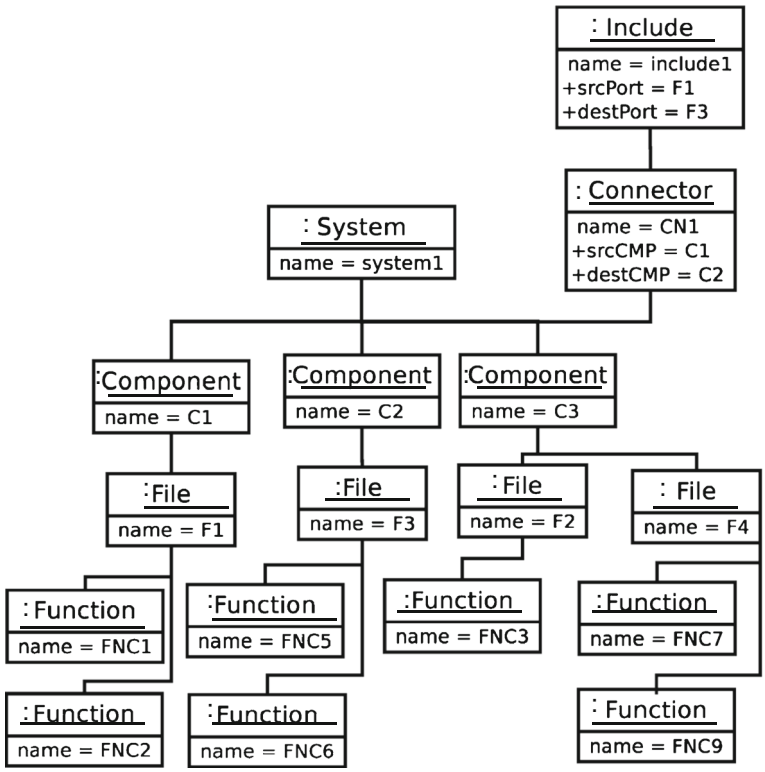

Fig. 7 Example Architecture 2

we have 6 matched Entities (F1, F3, FNC1, FNC2, FNC5, FNC6), 5 moved Entities (F2, FNC3, F4, FNC7), 1 added Entity (FNC9) and 2 deleted Entities (FNC4, FNC8).

Step 3 *Match Connectors*
The next step is to match the Connectors based on the Components that were previously matched. In this case, the two Connectors are matched since the corresponding Components have already been matched.

Step 4 *Match Bindings*
Finally, the two Bindings are matched since they are of the same type (i.e. "Include" Bindings), they belong to matched Connectors, and connect already matched Entities.

Therefore, in order to calculate the similarity score, we calculate the fraction in Eq. 1. In this case, the total number of elements is 18 and the matched elements are 12 and the distance score is computed as $\frac{18-12}{18} = \frac{1}{3}$. Once the architectures for all considered (e.g. procedural) systems S_1, S_2, ... S_9 for a given relation combination have been calculated, the distance value for a given system S_i will be normalized and represent the distance score between the architecture of system S_i being evaluated and the reference ground truth architecture A.

7 Ground Truth Architecture Validation Process

The ground truth validation process is based on answering the question as to whether or not the thirteen used relations are adequate for extracting the ground truth architecture of a software system. In order to avoid any subjective interpretations, we opted to focus and base our validation process on the analysis of five large software systems for which the ground truth has already been established as an authoritative one. These five systems are *Bash*, *ArchStudio*, *Itk*, *Hadoop*, and *Chromium*, the architectures of which have been analyzed by Lutellier et al. and presented in Lutellier et al. (2015). These ground truth architectures are also available at *http://asset.uwaterloo.ca/ArchRecovery*. However, the ground truth architectures for these systems are reported in the form of collections of clusters which contain source files. As our clusters may contain any source element (e.g. functions, methods, types, classes, structures, vatriables), we had to lift the elements of our clusters to the *file* they are *declared in* (if the entity pertains to a declaration), or to the *file* they are *used in* (if the entity is a reference).

More specifically, the ground truth validation process is composed of four steps.

Step 1 The system's source code is parsed, and an RSF file with all thirteen relations is produced.

Step 2 Each source code entity that appears in the RSF file is lifted to its corresponding file, and the RSF file now contains relations that appear between lifted files. Consequently the ACDC tool is called, and a system architecture $\mathcal{A}_S$ is extracted using all thirteen relations. The $\mathcal{A}_S$ architecture is represented by clusters that now contain only source code files. Also, let the reported authoritative ground truth architecture for system $\mathcal{S}$, be denoted as $\mathcal{A}_{Sgt}$ (Lutellier et al. 2015).

Step 3 As the granularity of the extracted architecture $\mathcal{A}_S$ by ACDC is much higher than the granularity of the reported ground truth archutecture $\mathcal{A}_{Sgt}$, we merge clusters in $\mathcal{A}_S$ by considering their overlap scores with clusters in $\mathcal{A}_{Sgt}$. Clusters in $\mathcal{A}_S$ that have the highest overlap with cliusters in $\mathcal{A}_S$ are merged. Once a cluster is merged with others, it is not considered again for future merges.

Step 4 In this final step, the ground truth architecture $\mathcal{A}_{Sgt}$ and the extracted architecture $\mathcal{A}_S$ are compared cluster per cluster, and a similarity score using Tversky's index, with parameters $\alpha = 1$ and $\beta = 0$ is computed. Here, the ground truth architecture $\mathcal{A}_{Sgt}$ is considered the prototype, and the architecture $\mathcal{A}_S$ the variant. The similarity score ranges from 0 to 1 and is computed based on the Eq. 3 given below:

$$S(gt_i, s_j) = \frac{|gt_i \cap s_j|}{|gt_i \cap s_j| + \alpha|gt_i - s_j| + \beta|s_j - gt_i|} \tag{3}$$

where, gt_i is the i^{th} cluster of the ground truth architecture, and s_j is the j^{th} cluster of the extracted architecture. The results are reported in Table 4.

More specifically, in Table 4 the results indicate that in all systems and all clusters involved, the average similarity of the clusters s_j in $\mathcal{A}_S$ that match with the clusters gt_i in $\mathcal{A}_{Sgt}$, is one or two orders of magnitude higher than the average similarity of the next best match. Table 4 indicates that for ArchStudio, the average similarity of the matched clusters is 0.7514 while the average similarity of the next best score drops to 0.0927. For Bash the similarity score is low at a value of 0.2143, and the average similarity of the next best sxore drops to 0.0690. Manual inspection revealed that the Bash system is relative small and the ground truth architecture reported by Lutellier et al. (2015) follows exactly the source

Table 4 Comparison results between extracted architecture and ground truth architecture

System	# of Clusters in Ground Truth (Lutellier et.al.)	# of Clusters in Extracted (13 relations)	Avg. Max Similarity (On all clusters)	# Avg. Next Best Match (On all clusters)
Bash	14	10	0.2143	0.069
ArchStudio	57	52	0.7514	0.0927
ITK	11	11	0.6492	0.0400
Hadoop	67	44	0.5	0.15
Chromium	44	44	0.5981	0.058

code's directory structure, while the clustering process takes into account more information extracted from the source code (i.e. the thirteen different relations).

For a more detailed view of these scores, consider the values obtained for ITK, depicted in Table 5, where Cluster #6019 in $\mathcal{A}_S$ matches best with cluster #6 in $\mathcal{A}_{Sgt}$ (score 0.9161). Looking at all other similarity scores of cluster #6019 with all other clusters in ITK's $\mathcal{A}_{Sgt}$, we clearly identify that when the max similarity score of #6019 is compared with any other score in any other cluster in $\mathcal{A}_{Sgt}$, it is two orders of magnitute higher. The same results are depicted for clusters #4907, and #948 in $\mathcal{A}_S$. We also observe the case of cluster #3197 which best matches with cluster #1, but cluster #3197 also matches (i.e. it splits) with $\mathcal{A}_{Sgt}$ cluster #6, and cluster #11. Overall, our results indicate that the thirteen relations we have used provide significant and strong evidence that the extracted architecture $\mathcal{A}_S$ aligns well with the clusters of the ground truth architecture $\mathcal{A}_{Sgt}$.

Similarly, Table 6 depicts the percent value of the SLOCs matched and the number of files matched for those clusters in $\mathcal{A}_{Sgt}$ and in $\mathcal{A}_S$ that exhibit max similarity. For example in Bash 34.5 % of SLOCs were matched among all matched clusters, while in ITK 73.1 % of files were matched, among all matched clusters, when the ground truth architecture and the extracted ITK architecture are compared.

Table 5 Tversky's Index similarity scores between clusters #3197, #4907, #6019, and #948 in $\mathcal{A}_S$ and the clusters reported in $\mathcal{A}_{Sgt}$

Cluster ID in $\mathcal{A}_{Sgt}$	Cluster #3197 in $\mathcal{A}_S$	Cluster #4907 in $\mathcal{A}_S$	Cluster #6019 in $\mathcal{A}_S$	Cluster #948 in $\mathcal{A}_S$
#1	0.3341	0	0.0064	0
#2	0.0315	0	0	0
#3	0.1454	0	0	0
#4	0.0880	0	0	0
#5	0.0722	0	0.9161	0.0067
#6	0.3097	0	0	9.55E-04
#7	0.0510	0	0	0
#8	0.0020	0	0	0
#9	0.1096	0.7772	0	0
#10	0.0900	0	0.0034	0
#11	0.2004	0	0.0043	0.8249

Table 6 Coverage percentage of matched clusters in terms of number of matched lines, and number of matched files

System	SLOC Coverage of Matched Clusters	File Coverage of Matched Clusters
Bash	34.5 %	31.8 %
ArchStudio	81.4 %	75.8 %
ITK	78.2 %	73.1 %
Hadoop	61.2 %	61.4 %
Chromium	62.7 %	62 %

8 Case Study

8.1 Case Study Infrastructure and Systems Under Analysis

An important consideration that arises is whether the underlying programming paradigm, affects in any way the process and the technique that should be used for software architecture recovery.

On one hand, object oriented systems are structured around classes that encapsulate methods and promote the concept of information hiding through the use of access specifiers such as public, private, or protected members. Furthermore, object oriented systems promote the concepts of inheritance, polymorphism, and overloading. All these concepts directly affect the organization and structure of object oriented source code as compared to the structure and organization of systems written in imperative procedural languages. For example, classes and methods that belong to the same inheritance hierarchy, or contain overloaded methods, or are declared as friends (in the case of C++), would make sense from a software engineering point of view to group together. On the other hand, procedural systems, tend to be structured using a top-down approach following a routine-subroutine architectural style. In the procedural paradigm encapsulation, overloading and inheritance are not available features to consider, while call relationships and data access relationships are more important.

In this respect, our motivation to consider two different sets of case studies, one for procedural and one for object oriented systems, stems from the question that arises as to whether these two different programming paradigms have also an impact on the type of relations that are most suitable for architectural extraction. We consider these two paradigms as the two most prevalent ones in legacy systems. However, it would be also interesting to consider as future work, the identification of relations that are most suitable for the architectural recovery of systems written in functional programming languages, or programs written in 4GLs, such as data reporting applications.

Our case study were conducted on six quad-core Intel Core i7 3,2 GHz machines with 8GB physical memory each, running Ubuntu Linux. The ACDC parameters were set to Body_Header, Subgraph_Dominator, and Orphan_Adoption. The max cluster size (not enforced though during the Orphan Adoption process) was set to 20 elements per cluster. The case study aimed to evaluate the impact different relation combinations have on the accuracy of the extracted architecture of a number of open source, large procedural and object oriented systems obtained from various fields, such as Operating Systems, Databases, Networks, and AI. More specifically, we considered 9 procedural and 6 object oriented systems. All the systems are desktop applications and do not involve any external database. A

possible future work would be to examine whether there is any difference in the obtained results, by considering the type or the characteristics of the application involved.

The procedural systems that we examined are:

Bash: The Unix shell and command language.
Clips: The software tool for building expert systems.
OpenSSH: The suite of security-related network-level utilities based on the SSH protocol.
OpenSSL: The open source implementation of the SSL and TLS protocols.
OpenVPN: The open source software application that implements virtual private network connections for secure point-to-point communication.
Freeglut: The open source alternative to the OpenGL Utility Toolkit (GLUT) library.
Putty: The free and open source terminal emulator, serial console and network file transfer application.
Tcsh: The Unix shell.
Zsh The Unix shell.

Specific information about the procedural systems such as Line of Code (LOC), years in operation, current version, number of relations included in the RSF file and, number of nodes in the RSF file, is presented in Table 7.

The object oriented systems we have examined are:

Apache Ant: The software tool for automating software compile/build systems.
Apache Maven: The software tool for automating software compile/build processes.
Apache Ivy: The Apache Ant sub-component that is used to resolve project dependencies.
jEdit: The open source software text editor.
jHotDraw: The Java framework for producing technical and structured Graphics.
Texmaker: The free and cross-platform LaTeX editor for Linux, OS X and Windows systems.

Specific information about the object oriented systems we have experimented with, such as Line of Code (LOC), years in operation, current version, number of relations included in the RSF file and, number of nodes in the RSF file, is presented in Table 8.

Table 9 presents the proportion of instances of the different relations as these appear in the different RSF files. More frequent relations (i.e. with higher proportion values) tend to appear in the "good" relations set, but this is not always the case. For example, the *Includes* relation has low average proportion value (2 %) and has been found to be a good relation

Table 7 Procedural Systems under Examination

System	LOC	Years in Operation	Current Version	RSF Lines	No of Nodes
Bash	99,871	17	7.0	22,850	7,246
Clips	91,021	21	6.3	31,217	6,752
Freeglut	22,832	17	3.0.0	9,275	4,161
OpenSSH	63,999	17	7.1	19,370	4,037
OpenSSL	298,767	18	1.0.2h	62,283	23,385
OpenVPN	61,606	15	2.3.1	18,876	4,411
Putty	85,716	17	0.67	18,439	6,111
Tcsh	52,143	23	6.19	12,995	3,138
ZSH	98,061	26	5.2	16,905	5,309

Table 8 Object Oriented Systems under Examination

System	LOC	Years in Operation	Current Version	RSF Lines	No of Nodes
ApacheAnt	107,243	16	1.9.7	63,332	18,149
ApacheIvy	72,724	12	2.4.0	48,184	11,280
Apache Maven	78,442	12	3.3.9	29,827	11,506
jEdit	118,491	18	5.3.0	49,903	12,875
JHotDraw	80,160	20	7.2	33,797	10,712
TexMaker	59,434	13	4.5	18,643	3,970

to use (see results in following sections). These results indicate that the density of relations may play a role in the clustering process, but on the other hand these high density relations, are also the relations we have always suspected and considered as software engineers to be the important ones (i.e. calls, accesses, sets), because they facilitate the flow of data and control information accross the system.

8.2 Analysis Results

In this section, we present the results of our case study. This study aims to address whether there are combinations of relations that can be easily extracted from the source code and at the same time can be used to provide through clustering, an accurate depiction of a system's architecture. Another question is whether there are combinations of relations the omission of which will hinder the obtained result. Finally, we would like to evaluate whether there is a difference in the obtained results when procedural or object oriented systems are considered. For each selected relation combination we provide an indication of the ease of extraction of this combination. More specifically, by the symbol ++ we denote the easiest to extract relation combination and by – the most difficult one. The up arrows and down arrows indicate

Table 9 Relation density values for different relation entries in RSF files

Relation	Average density value	Standard Dev.
Calls	0.13	0.05
Includes	0.02	0.01
Sets	0.06	0.01
Accesses	0.17	0.02
Class Belongs To File	0.006	0.002
Inherits From	0.003	0.002
Has Type	0.002	0.001
Defined In	0.07	0.01
Declared In	0.1	0.02
Attribute Belongs To Class	0.04	0.01
Method Belongs To Class	0.09	0.01
Accessible Entity Belongs To File	0.05	0.06
Uses Type	0.02	0.01

intemediate levels between two adjacent classifications (e.g. between ++ and + classification). In this respect, the +↓ would be the least possible acceptable combination, based on ease of relation extraction.

8.2.1 Procedural Systems

The results obtained for the procedural systems examined are depicted in Figs. 8 and 9, Tables 10, 11 and 12. Even though we report average difference scores, the obtained results are consistent for all nine procedural systems we have considered.

Figure 8 presents the average distance score for each combination from all the 9 procedural systems we have considered. The X-axis presents the relation combinations as these are generated in order (i.e. 1-relation combination, all 2-relation combinations, all 3-relation combinations, to 9-relation combinations). For more than 9 relations the distance drops close to 0 and these are not depicted in X-axis of Fig. 8. Similarly, the Y-axis presents the distance score between the system's extracted architecture, and the ground truth's architecture. As it can be observed, three groups of points emerge. First, points that relate to a score above the 0.6 value are considered bad combinations because the distance between the extracted architecture instance model and the ground truth reference model is high. Then, there are combinations, with average distance scores between 0.3 and 0.6. Those combinations represent instances that are closer to the ground truth, but still are not sufficient. Finally, there are the combinations with distance score below 0.3. These combinations are very close to the ground truth and are considered good combinations that can be used for architectural extraction.

Figure 9 depicts similar information as Fig. 8, however here the distance score data in the Y-axis are sorted in ascending order. This depiction of the results indicates two major step points, marked with the two vertical lines. The first "step" point separates the combinations

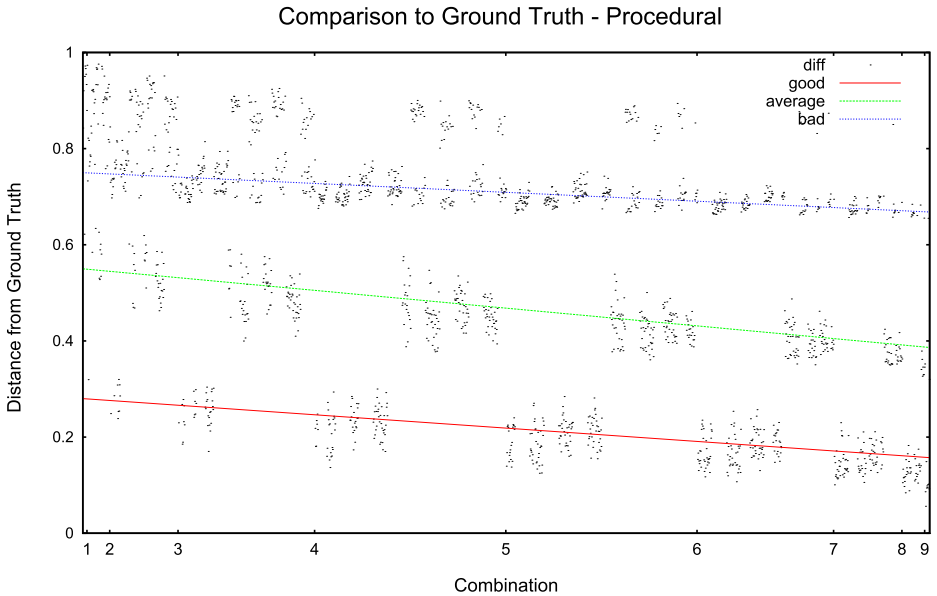

Fig. 8 Average distance score for Procedural Systems. X-axis relation combination generated orderly

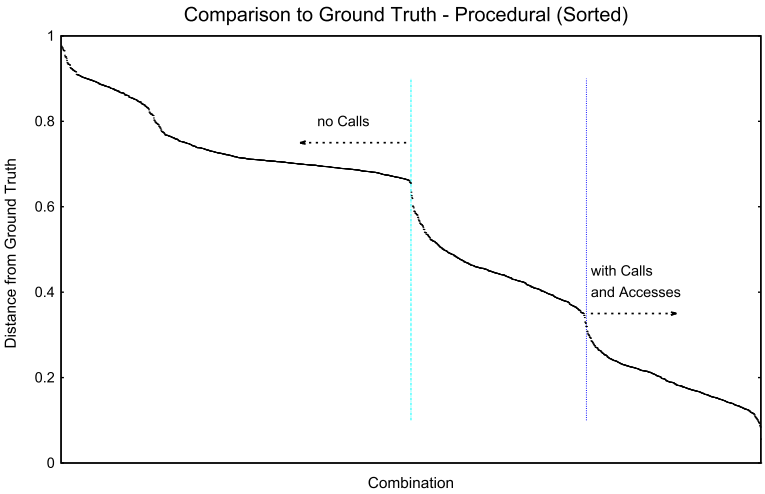

Fig. 9 Sorted average distance score for Procedural Systems. X-axis relation combination selected for sorted depiction of Y-axis distance result

that do not have the relation *Calls* (left) from those that have it (right). The second "step" point separates the combinations that do not have the relations *Accesses* or *Calls* (left) from those that have them both (right).

Table 10 Best combinations of relations - Procedural Systems

	Combination	Diff Score	Z Score < -2 p Value $= 0$	Ease of Relation Extraction
1	Calls	0.622	−2.72	++
2	Calls, Accesses	0.319	−3.43	+
	Calls, Defined In	0.528	−2.00	+
3	Accesses, Calls, Accessible Entity Belongs to File	0.249	−2.76	+ ↑
	Accesses, Calls, Class Belongs to File	0.308	−2.42	+ ↑
	Accesses, Calls, Declared In	0.238	−2.82	-
	Accesses, Calls, Includes	0.254	−2.73	+ ↑
	Accesses, Calls, Has Type	0.319	−2.35	- ↓
	Accesses, Calls, Defined In	0.253	−2.73	-
	Accesses, Calls, Set Variable	0.309	−2.42	+ ↓
	Accesses, Calls, Attribute Belongs to Class	0.286	−2.54	+ ↑
	Accesses, Calls, Uses Type	0.286	−2.55	-↓
4	Accesses, Calls, Declared In, Includes	0.170	−2.46	+ ↓
	Accesses, Calls, Accessible Entity Belongs to File, Declared In	0.185	−2.38	+↓
	Accesses, Calls, Accessible Entity Belongs to File, Defined In	0.189	−2.36	+ ↓
5	Accesses, Calls, Accessible Entity Belongs to File, Declared In, Includes	0.137	−2.06	+

Table 11 Worst combinations of relations - Procedural Systems

	Combination	Diff Score	Z Score	p Value	Ease of Relation Extraction
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, Includes	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, Includes	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, Includes	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 10 depicts the best relation combinationss. 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

Table 12 Average and Standard Deviation with and without a combination of relations - Procedural Systems

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

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. As far as the p Value is concerned, we conducted a Chi Square Test in order to reveal the significance level of our results. Our Null Hypothesis was that each relation combjnation and the corresponding obtained outcome (good, average, bad) is independent. As we can see, for all combinations that energe as been among the best, the Null Hypothesis is rejected with a probability of almost 100 %, as p Values are very close or equal to 0, and therefore, we can conclude that there is a connection between the best combinations and the obtained outcome.

On the other hand, Table 11 depicts the worst relations that can be used for architectural extraction as observed in our case study. These results indicate that these relations when used on their own (i.e. not combined with other relations) do not consitute a good choice for architectural extraction.

Finally, Table 12, presents the mean distance score and standard deviations for all the cases that contain a specific combination, as well as for all the cases that do not contain it. The fact that when a specific 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, indicates that this combination is important for architecture recovery, and therefore should be always considered.

8.2.2 Interpretation

In this section we interpret the results presented above, by addressing the following related questions.

Are there any relations that are the most important and should be always used when available?

Our case study indicated that there are two relations whose presence makes significant difference. First of all, the *Calls* relation seems to be the most important relation in procedural systems. As we can see from Fig. 8 there are three groups of distance scores, which are very clearly separated. The difference between the top group and the other two groups is the presence of the *Calls* relation. This means that if we choose not to include the *Calls* relation in our set of relations that will be used to extract a system's architecture, then we cannot obtain a distance score less than 0.6, which is not a satisfactory outcome. It is therefore safe to assume that the *Calls* relation should always be used, even though it is not the easiest relation to extract.

This is also visible in Table 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 *Calls* relation. When in the considered combinations the *Calls* 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 one has to pay if decides not to use the *Calls* relation. Even if all other relations are used, except *Calls*, the distance score is 0.66, which is higher than the distance score containing at least *Calls*. Therefore, our proposal is to start with the *Calls* 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 *Calls*, another relation that is recommended to be part of the set of relations that will be used is the *Accesses* relation. This relation combined with the *Calls* relation makes a good choice, as depicted in Table 10. It is noted that these two relations are present in all the best combinations and are absent from all the worst combinations (see Table 11). Furthermore, the pair *Calls, Accesses* is what makes the difference between the average and good relations in Fig. 8. All the combinations below 0.3, which are considered good combinations for recovery, contain both *Calls* and *Accesses* relations. Furthermore, the combinations that contain these two relations have a mean value of 0.189 and a standard deviation of 0.094 compared to 0.796 and 0.139 respectively for those combinations that do not contain them. As a result, our proposal is that *Calls* and *Accesses* should always be considered for architecture recovery.

Are there any relations that do not offer any apparent value?

In our case study we have encountered a number of relations for which their absence or presence did not significantly affect the quality of the extracted architecture as this is measured by its distance to the ground truth reference standard architecture. These relations are *Has Type*, *Uses Type* and, *Sets*. As it is depicted in Table 10, when these relations are added to the *Calls-Accesses* combination the distance score of 0.319 we have already obtained by *Calls-Accesses*, is only slightly enhanced. The same happens if we add these relations to any combination. Given the fact that for these relations their contribution is not significant,

it is safe to conclude that they can be omitted for the sake of faster processing, with no significant loss of accuracy.

Are there any relations that should not be used, or not be used on their own?

Throughout our case study we did not encounter any relation that when it was added to a combination, has negatively affected the similarity score.

Additionally, as depicted in Table 11, combinations containing information only about structural aspects of the system, such as *Attribute Belongs to Class*, *Has Type Class Belongs to File* and *Accessible Entity Belongs to File*, *Includes*, *Defined In*, *Declared In* without been combined with any relation that encompasses data or control flow information, tend to produce a non acceptable result. However, when these relations are combined with data and control flow relations such as *Calls* and *Accesses* produce the best results. Our case study indicated that these relations should be avoided to be used on their own without been considered in combination with a relation that defines connections between these entities such as *Calls* and *Accesses*.

How many relations do we need for accurate architecture recovery?

The answer to this question depends on which relations one has available to use, first of all. For example, if for some reason the *Calls* relation is not available, even if the other twelve relations are used the distance score is about 0.65 which is very high. Our case study indicated that *four* well chosen relations are adequate for an accurate architecture extraction in procedural systems. More specifically, these top relations include *Calls* and *Accesses* followed by *Declared In*, *Includes*,*Accessible Entity Belongs to File*, and *Defined In*. In this context, the structure distance score between the extracted architecture and the one considered as the reference one using four relations is 0.170, which indicates a very accurate result. When five of these relations are used the average distance drops to 0.137. For more than 6 relations, because the amount of the information that is provided is close to the total amount of information of the system, provided that *Calls* and *Accesses* are included, the difference between the distance scores in the various combinations is non-significant (in the range 0.09 - 0.15). As a result, for more than 6 relations, if *Calls* and *Accesses* are included, the choice of the other relations does not really matter.

8.3 Object Oriented Systems

Similar to procedural systems, a case study was also conducted on object oriented systems for the purpose of assessing whether the results obtained for procedural systems also hold or not, and if not, what are the notable differences. The obtained results for the object oriented systems are presented on Figs. 10 and 11, Tables 13, 14 and 15. These tables present average difference scores, but the obtained results are consistent for all six object oriented systems we have considered in this study.

More specifically, Fig. 10 presents the average distance score for each combination applied on all systems examined. As it is observed, there are four groups of points, those above 0.8 which are considered bad combinations because the similarity between the two architecture instances is extremely low, those between 0.8 and 0.6, which are still very different from the ground truth, those between 0.3 and 0.5 which are considered acceptably similar and finally, those with distance score below 0.3 which indicates architectures that are very close to the ground truth and correspond to the best combinations to be used for architectural extraction according to the obtained results.

Similarly, Fig. 11 depicts the same information as Fig. 10, with the difference that data in the Y-axis are now sorted in ascending order based on their distance score. As it is depicted, there are three steps identified in the graph, marked with the three corresponding vertical

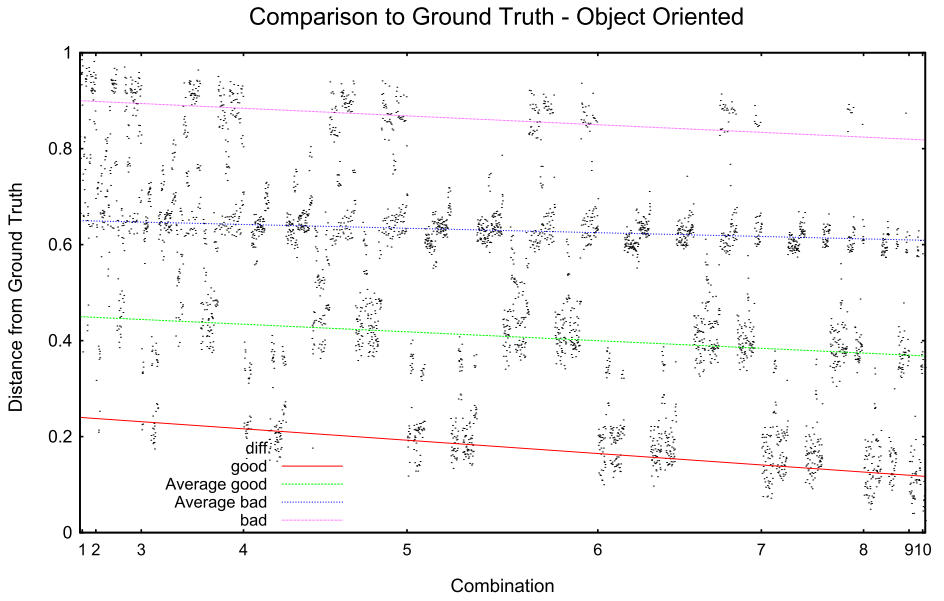

Fig. 10 Average distance score for object oriented Systems. X-axis relation combination generated orderly

lines. The first line separates the combinations that do not have the relations *Calls*, *Accesses* or *Sets* (left) from those that have at least one of them (right). The second line separates the combinations that do not have the *Calls* relation (left) from those that do (right). Finally, the third line separates the combinations that do not have the relations *Accesses* or *Calsl* (left) from those that have them both (right).

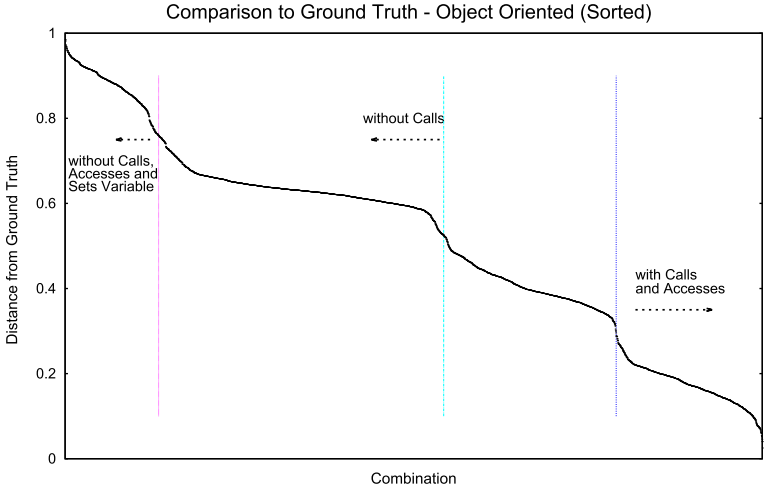

Fig. 11 Sorted average distance score for object oriented Systems. X-axis relation combination selected for sorted depiction of Y-axis distance result

Table 13 Best combinations of relations - object oriented systems

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

Table 13 depicts the best relation combinations obtained for object oriented systems. For each combination, the average Distance Score for all systems, the Z score as well as, the p Value of a Chi Square Test, is presented. Z Score represents how many standard deviation

Table 14 Worst combinations of relations - object oriented systems

	Combination	Diff Score	Z Score < -2 p Value > 0.5	Ease of Relation Extraction
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, Includes	0.950	1.61	+↓
	Attribute Belongs to Class Has Type, Includes, 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, Includes, 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, Includes, 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, Includes, Uses Type	0.935	1.84	–
	Attribute Belongs to Class, Declared In, Defined In, Has Type, Inherits From	0.935	1.84	-↓

Table 15 Average and Standard Deviation with and without a combination of relations - object oriented systems

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

points the given distance value is from the mean of distance obtained from all systems being examined, for this relation combination. A Z score less than -2 represents that the value is in the top 0.5 % of this set. Values that are far from the mean, correspond to combinations that are considered the best ones. As far as the p Value is concerned, we conducted a Chi Square Test in order to reveal the significance level of our results. As with the case in procedural systems, our Null Hypothesis was that each relation combination and the obtained outcome (good, average good, average bad, bad) is independent. As we can see, for all the combinations that considerd the best, the Null Hypothesis is rejected and with a probability of almost 100 % and therefore, we can conclude that there is a connection between the best combinations and the obtained result.

On the other hand, Table 14 depicts the worst relation combinations, according to our case study, for object oriented systems. As with Table 13, this table also provides information on Distance Score, the Z Score as well as, the p Value for the Chi Square Test of each combination. We can see here, that all these relations do not provide acceptable results and therefore, their use for architectural extraction is not suggested.

Finally, Table 15, depicts the mean distance score and the standard deviation for all cases that contain a combination, as well as for all the cases that do not contain it. The fact that when a relation combination is present, we have a low distance score and standard deviation, while when this combination is absent the distance score rises significantly, indicates that this combination is important for architecture recovery and should be always considered for architecture recovery in object oriented systems.

8.3.1 Interpretation

As with the case of procedural systems, we interpret the obtained results by answering a set of related questions.

Are there any relations that are the most important and should be always used when available?

As was discussed in the previous sections regarding procedural systems, the relations that have the highest impact for architecture recovery were the *Calls* and *Accesses* relations. Although, in object oriented systems these relations are still identified as important, these are best to be combined with relations related to the overall organization of the source code. As it is depicted in Table 15 relations that reveal the organization of the classes in object oriented systems provide significant information for accurate architecture extraction (distance from the ground truth less than 0.2), especially when combined with one or more relations about the interaction of the system's elements (e.g. *Calls* and *Accesses*).

Furthermore, in Fig. 11 a major difference is observed when compared to the corresponding Figure for procedural systems. More specifically, a large drop is observed in the beginning of the diagram, separating the combinations that do not contain any of the *Calls*, *Accesses* and *Sets* relations, from those combinations that they do contain these relations. As a result, we can infer that the absence of information on interactions between Components results to noisy architecrure extraction manifested by high distance scores of the obtained architecture, from the one considered as the reference standard. The obtained results indicate that in order to acquire an architecture close to the reference standard we need at least three relations, namely the *Calls* relation, *Accesses* relation, and a relation such as *Method Belongs to Class*, *Declared In* or *Defined In* that relate to the overall organization of the source code. With only these three relations the distance score is about 0.25, which can be considered satisfactory. The results indicate that when these are combined with additional code structure related relations such as *Attribute Belongs to Class* or *Class Belongs to File* an even more accurate depiction of the architecture can be extracted with distance scores less than 0.2 if four of these relations are combined, or 0.15 if five of these relations are combined.

Finally, an interesting point regarding object oriented systems is that source code structure-related relations are easy to extract, as indicated in Table 3. Such relations can be extracted with a simple scanning of the source code, and therefore do not require extra parsing, processing, or linking effort. If these relations are ignored, as Table 15 depicts, the average distance score rises to values higher than 0.7, which actually indicates that the remaining relations are not very useful for architectural extraction. Therefore, for architecture extraction in object oriented systems is best to start with the combination *Calls*, *Accesses* and one of the following *Declared In*, *Defined In* or *Method Belongs to Class*, and consider more relations, if more accuracy is required and more resources can be allocated for the extraction of such relations from the source code.

Are there any relations that do not offer any apparent value?

Our case study revealed that there are some relation combinations that do not have a positive or negative impact for architecture extraction. These relations are *Has Type*, *Inherits From* and *Uses Type*. As it is depicted in Table 13, when these relations are added to the *Calls, Accesses* combination the distance score is only slightly changed. The same happens if we add these relations to any combination. In this respect, we can infer that these relations can be omitted altogether, for the sake of faster processing, with no loss in the accuracy of the extracted architecture.

Are there any relations that should not be used, or not used on their own?

As with procedural systems, throughout our case study we did not encounter any relation that when it was considered it had negatively affected the similarity score. However, there were a number of structure-related relations that when not combined with data or control flow, produce a non accurate depiction of the system architecture.

Table 11 depicts relation combinations containing information about structural aspects of the system, such as *Attribute Belongs to Class*, *Class Belongs to File* and, *Method Belongs to Class*, which when combined with information related to data or control flow (e.g. *Calls*, *Accesses*), produce very accurate results. For this reason, these relations are better to be avoided from being used on their own for architecture recovery without considering at the same time data or control flow related relations, such as *Calls*, *Accesses*, or *Sets* relation.

How many relations do we need for accurate architecture recovery in object oriented systems?

The results indicate that for object oriented systems four relations are the minimum for an accurate extraction (i.e. drop on a distance score below or around 0.17), with the best combination being the use of *Calls*, *Accesses* pair with the *Defined In* and *Method Belongs to Class* relations. For better and more sound results, five relations are best to be used.

These include the use of *Calls*, *Accesses* pair, and the addition of a relation such as *Method Belongs to Class*, *Includes*, *Attribute Belongs to Class*, or *Class Belongs to File* provides a distance score lower than 0.15, which indicates an accurate extraction. For more than five relations, because the amount of the information that is included is close to the total amount of information provided by all available relations, provided that both structural and relation about interaction of Components are included as discussed above, the difference between the distance scores of the combinations are insignificant (in the range 0.07 - 0.15). As a result, the selection of more than the appropriate five relations does not really contribute to the accuracy of the extracted architecture.

9 Conclusion and Future Work

In this paper, we presented the results of a case study for assessing the impact different source code relations have on the accuracy of clustering-based architecture extraction. Clustering-based architecture extraction is a well investigated subject in the area of reverse engineering, and a large number of techniques have been proposed in the research literature. However, there is limited work on the identification of which relation combinations can be easily obtained from the source code, and yet have a high impact on extracting an accurate depiction of a system's actual architecture. For this purpose, we have designed and implemented a case study in order to evaluate which relations have the highest positive impact on cluster-based architectural extraction. Furthermore, we are interested in classifying these relations according to their ease of extraction from the source code, by considering

as a factor whether they require the use of simple scanners, and simple parsers, or require specialized parsers and linkers. Our case studies were differentiated and applied first on large open-source procedural systems, and second on large open-source object oriented systems.

Our results indicate that the selection of the relations has a significant impact on the accuracy of the extracted architecture, as this is compared to a reference architecture used as a ground truth. Depending on the accuracy sought, our case study indicated that we can extract an accurate architecture by using a small number of selected relations. More specifically, in procedural systems, relations related to data and control flow between Components, such as the "Call" and "Access" relations, should be always used and never omitted. As new relations are added, the extracted architecture becomes more accurate. However, a quite accurate architecture can be extracted by a combination of four relations that utilizes *Calls* and *Accesses* relations, followed by a combination of *Declared In*, *Includes*, *Accessible Entity Belongs to File*, or *Defined In*.

In object oriented systems, the organization of the source code (class-subclass hierarchies, and interfaces) reveals much information about the architecture when combined with data or control-flow related information. As expected, the combinations that have the greater number of relations produce the most accurate results. However, when the relations that are considered the most important are missing, the similarity to the ground truth architecture deteriorates dramatically. For object oriented systems, a minimum of four relations can be used (*Calls*, *Accesses* pair, combined with *Method Belongs to Class*, *Includes*), but for more accurate and stable results five relations are best to be used. These include the *Calls*, *Accesses* pair, combined with *Method Belongs to Class*, *Includes Attribute Belongs to Class*, *Class Belongs to File*, *Declared In*, or *Defined In* relations.

Our case study indicates that there were no relations that have a negative impact on the obtained result and should be consequently avoided. However, certain relations were found whose use should only be considered in combination with the *Calls* and *Accesses* relations, otherwise they do not make any significant difference if used only on their own. These include *Attribute Belongs to Class*, *HasType Class Belongs to File* and *Accessible Entity Belongs to File*, *Includes*, *Defined In*, *Declared In* for procedural systems, and *Attribute Belongs to Class*, *Class Belongs to File*, *Method Belongs to Class* and *Uses Type* for object oriented systems. It is noted though that even these relations are not valuable on their own, they provide the best results when combined with *Calls* and *Accesses* relations.

Overall, we believe that this work covers an area that is of practical value to both researchers and practitioners in the reverse engineering community, and sets the stage for further experimentation on this subject. One such area is to experiment with systems written in other popular programming languages such as Python which supports multiple programming paradigms (object oriented, imperative, functional, procedural), and investigate whether there is any difference on the relations that need be considered for architecture recovery from the ones we have identified in this study.

Another area is to assess whether the application domain has an impact on the obtained results, and whether other relation combinations are better to use than the ones identified.

Finally, it is interesting to experiment with distributed systems and consider relations involving run-time information and message exchange patterns, by assuming that these messages may hide valuable information about the architecture of the system.

Acknowledgments We would like to thank Stergios Ientsek for his contribution related to enhancements of the extraction tools and the production of RSF files, and the anonymous reviewers for their constructive comments.

References

Adams B, Tromp W, De Meuter H, Hassan A (2009) Can we refactor conditional compilation into aspects? In: Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development, AOSD 09, pages 243–254, New York, ACM

Akers RL, Baxter ID, Mehlich M, Ellis B, Luecke K (2005) C++ component model reengineering by automatic transformation. In: CrossTalk, The Journal of Defense Software Engineering

Allen R (1997) A formal approach to software architecture, Ph.D. thesis, Carnegie Mellon School of Computer Science

Andritsos P, Tzerpos V (2005) Information-theoretic software clustering. IEEE Trans Softw Eng 150–165

Anquetil N, Lethbridge T (1998) Extracting concepts from file names: a new file clustering criterion. In: Proceedings of the international conference on software engineering. Association of Computing Machinery (ACM) Press, pp 84–93

Bass L, Clements P, Kazman R (2012) Software architecture in practice. Addison-Wesley Professional, 3rd Edn

Bass L, Clements P, Kazman R (2013) Software architecture in practice. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

Bauer M, Trifu M (2004) Architecture-aware adaptive clustering of oo systems. In: Proceedings of the conference on software maintenance and reengineering. IEEE Computer Society Press, pp 3–12

Bois BD et al. (2007) Supporting reengineering scenarios with FETCH: an experience report. ECEASST 8

Bojic D, Velasevic D (2000) A use-case driven method of architecture recovery for program understanding and reuse reengineering. In: IEEE Conference on Software Maintenance and Reengineering, CSMR'00. pp 23–33

Boughanmi F (2010) Multi-language and heterogeneously-licensed software analysis. In: Proceedings of 17th working conference on reverse engineering. pp 293–296

Bowman IT, Holt R (1998) Software architecture recovery using conway's law. In: Proceedings of the 1998 conference of the centre for advanced studies on collaborative research. CASCON '98, 6 IBM Press

Canfora G, Czeranski J, Koschke R (2000) Revisiting the delta-ic approach to component recovery. In: Proceedings of the working conference on reverse engineering. IEEE Computer Society Press

Chiricota Y, Jourdan Y, Melanon F (2003) G. Software Components capture using graph clustering. In: Proceedings of the workshop on program comprehension. IEEE Computer Society Press, pp 217–226

Corazza A et al. (2011) Investigating the use of lexical information for software system clustering, CSMR, IEEE Computer Society 35–44

DeBaud JM, Moopen B, Rugaber S (1994) Domain analysis and reverse engineering. ICSM IEEE Comput Soc 326–335

Ducasse S, Pollet D (2009) Software architecture reconstruction: A process-oriented taxonomy. IEEE Trans Softw Eng 99(1)

Ducasse S, Tichelaar S (2003) Dimensions of reengineering environment infrastructures. Int J Softw Maint Res Pract 15:345–373

Feiler PH (2014) AADL and model-based engineering. Ada Lett ACM 34(3):17–18

Fischer M, Pinzger M, Gall H (2003) Analyzing and relating bug report data for feature tracking. In: Proceedings of the 10th working conference on reverse engineering, WCRE '03. IEEE Computer Society, Washington, pp. 90–,

Fleck G et al. (2016) Experience report on building astm based tools for multi-language reverse engineering. In: Proceedings of 23rd conference on software analysis, evolution, and reengineering, pp 283–687

Garcia J et al. (2011) Enhancing architectural recovery using concerns. In: Proceedings of the 2011 26th IEEE/ACM international conference on automated software engineering, ASE '11, IEEE, pp 552–555

Garcia J, Popescu D, Mattmann C, Medvidovic N, Cai Y (2011) Enhancing architectural recovery using concerns, 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11. IEEE 552–555

Garcia J, Ivkovic I, Medvidovic N (2013) A comparative analysis of software architecture recovery techniques, 28th International Conference on Automated Software Engineering, ASE 2013. IEEE 486–496

Garcia J, Krka I, Mattmann C, Medvidovic N (2013) Obtaining ground-truth software architectures. In: Proceedings of the 2013 international conference on software engineering. ICSE '13 IEEE Press, pp 901–910

Garlan D, Monroe RT, Wile D (1997) Acme: An architecture description interchange language. In: Proceedings of CASCON'97, Toronto, Ontario, pp 169–183

Imber M (1991) The CASE data interchange format (CDIF) standards. In: Long, F (ed) Software engineering environments, Ellis Horwood. pp 457–474

Jackson D (2012) Software Abstractions: logic, language, and analysis MIT press
Jerding D, Rugaber S (2000) Using visualization for architectural localization and extraction. Sci Comput Program:267–284
Kobayashi K et al. (2012) Feature-gathering dependency-based software clustering using dedication and modularity. In: Proceedings of the 28th international conference on software maintenance. IEEE Computer Society, pp 462–471
Koschke R, Canfora G, Czeranski J (2006) Revisiting the approach to component recovery, Science of Computer Programming, Special Issue on Software Analysis, Evolution and Re-engineering, pp 171–188
Kruchten P (1995) The 4+1 view model of architecture. IEEE Softw 12(6):42–50
Kuhn HW (1955) The Hungarian method for the assignment problem. Nav Res Logist Q:83–97
Lethbridge T, Tichelaar S, Ldereder E (2004) The dagstuhl middle metamodel: A schema for reverse engineering. Electr Notes Theor Comput Sci 94:7–18
Lung CH (1998) Software architecture recovery and restructuring through clustering techniques. In: Proceedings of the third international workshop on software architecture, ISAW '98, ACM, pp 101–104
Lung C-H (1998) Software architecture recovery and restructuring through clustering techniques. In: Proceedings of the Third International Workshop on Software Architecture. Association of Computing Machinery (ACM) Press, pp 101–104
Lungu M, Lanza M, Nierstrasz O (2014) Evolutionary and collaborative software architecture recovery with softwarenaut. Sci Comput Program 79:204–223. In: Proceedings of the Conference on Software Maintenance and Reengineering, CSMR '00, pages 23–, Washington, DC, USA, 2000 IEEE Computer Society
Lutellier T, Chollak D, Garcia J, Tan L, Rayside D, Medvidović N, Kroeger R (2015) Comparing software architecture recovery techniques using accurate dependencies. In: Proceedings of the 37th international conference on software engineering - vol 2, ICSE '15. IEEE Press, Piscataway, pp 69–78
Mancoridis S et al. (1999) Bunch: A clustering tool for the recovery and maintenance of software system structures. In: Proceedings IEEE international conference on software maintenance. IEEE Computer Society Press, pp 50–59
Mancoridis S, Holt RC (1996) Recovering the structure of software systems using tube graph interconnection clustering. In: Proceedings of international conference on software maintenance 1996. IEEE, pp 23–32
Mahdavi K, Harman M, Hierons RM (2003) A multiple hill climbing approach to software module clustering. In: Proceedings of the international conference on software maintenance, september. IEEE Computer Society Press, pp 315–324
Maqbool O, Babri HA (2004) The weighted combined algorithm: A linkage algorithm for software clustering. In: Proceedings of the conference on software maintenance and reengineering. IEEE Computer Society Press, pp 15–24
Maqbool O, Babri H (2007) Hierarchical clustering for software architecture recovery. IEEE Trans Softw Eng IEEE:759–780
Medvidovic N (1995) Formal definition of the chiron-2 software architectural style
Mendonça NC, Kramer J (2001) An approach for recovering distributed system architectures, Automated Software Engg, Kluwer Academic Publishers, pp 311–354
Muller H, Wong K, Tilley S (1992) A reverse engineering environment based on spatial and visual software interconnection models, ACM SIGSOFT Symposium on Software Development Environments. Association of Computing Machinery (ACM) Press, pp 88–98
Murphy GC, Notkin D, models KS (1995) Software reflexion Bridging the gap between source and high-level models. SIGSOFT Softw Eng Notes 20(4):18–28
Overbey JL, Johnson RE (2008) Generating rewritable abstract syntax trees. In: Gaševic D, Lämmel R, Wyk EV (eds) Software language engineering: first international conference (SLE 2008), Vol. 5452 of Lecture Notes in Computer Science. Springer, Berlin, pp 114–133
Patel C, Hamou-Lhadj A, Rilling J (2009) Software clustering using dynamic analysis and static dependencies. In: 13th European conference on software maintenance and reengineering, CSMR 2009, Architecture-Centric Maintenance of Large-SCale Software Systems, Kaiserslautern Germany, 24-27 March 2009, pp 27–36
Pinzger M et al. (2004) Architecture recovery for product families. In: van der Linden F (ed) Software product-family engineering, Lecture notes in computer science, vol 3014. Springer, Berlin, pp 332–351
Rayside D, Reuss S, Hedges E, Kontogiannis K (2000) The effect of call graph construction algorithms for object-oriented programs on automatic clustering. In: Proceedings workshop on program comprehension, 2000. Proceedings. IWPC, pp 191–200

Sartipi K, Kontogiannis K (2001) A graph pattern matching approach to software architecture recovery. In: Proceedings of the IEEE international conference on software maintenance (ICSM'01). IEEE Computer Society, pp 408–419

Sartipi K, Kontogiannis K (2003) A user-assisted approach to component clustering. J Softw Maint Evol Res Pract 15(4):265–295

Tilley S, Weiderman N, Woods S, Bergey J, Smith D (1999) Why reengineering projects fail. In: TECHNICAL REPORT CMU SEI99TR010 ESCTR99010. Software Engineering Institute

Tzerpos V, Holt RC (1996) A hybrid process for recovering software architecture CASCON

Tzerpos V, Holt RC (2000a) ACDC: An algorithm for comprehension-driven clustering. In: Proceedings of the seventh working conference on reverse engineering. IEEE, pp 258–267

Tzerpos V, Holt RC (2000b) On the stability of software clustering algorithms, 8th international workshop on program comprehension (IWPC 2000). IEEE:211–218

van Deursen A, Kuipers T (1999) Identifying objects using cluster and concept analysis. In: Proceedings of the international conference on software engineering. Association of Computing Machinery (ACM) Press, pp 246–255

Van Rompaey B et al. (2009) SERIOUS: software evolution, refactoring, improvement of operational and usable systems. In: CSMR. IEEE Computer Society, pp 277–280

Van Rompaey B, Demeyer S (2009) Establishing traceability links between unit test cases and units under test. In: 13th European conference on software maintenance and reengineering, CSMR 2009, Architecture-Centric Maintenance of Large-SCale Software Systems. Kaiserslautern Germany, pp 209–218

Vasconcelos A, Werner C (2004) Software architecture recovery based on dynamic analysis. In: XVIII Brazilian symposium on software engineering. Workshop on Modern Software Maintenance

Wilson RJ (1986) Introduction to graph theory. Wiley

Wu J, Hassan AE, Holt RC (2005) Comparison of clustering algorithms in the context of software evolution. ICSM IEEE Comput Soc:525–535

Wu J, Hassan AE, Holt RC (2005) Comparison of clustering algorithms in the context of software evolution. In: Proceedings of the 21st IEEE international conference on software maintenance. ICSM '05 IEEE Computer Society, pp 525–535

Xing Z, Stroulia E (2005) UMLDiff: An Algorithm for Object-oriented Design Differencing. In: Proceedings of 20th international conference on automated software engineering. ACM, pp 54–65

Zou Y, Kontogiannis K (2001) Towards a portable xml-based source code representation. In: Proceedings of international conference on software engineering (ICSE) 2001 workshops of XML technologies and software engineering (XSE), Toronto, Canada

Ioanna Stavropoulou received a B.Eng. in Electrical and Computer Engineering from the National Technical University of Athens with a major in Software Engineering. Ioanna is a second year M.Sc. candidate at Computer Science Department University of Toronto and she is currently working in the areas of model checking, feature interaction and modular composition in the context of product lines.

Marios Grigoriou has received B.Eng, in Electrical and Computer Engineering from the National Technical University of Athens, with a major in Software Engineering. He is currently working as a Data Analyst in the Hellenic Army. His research interests include software evolution, bug localization, and Artificial Intelligence.

Kostas Kontogiannis, is a Professor at Computer Science Department at Western University, where he holds a Western Research Chair in Software Engineering for Cyber-Physical Systems. Prior to joining Western, Kostas has served as a tenured faculty at the National Technical University of Athens, and at the University of Waterloo. Kostas has received a B.Sc. in Mathematics from the University of Patras, Greece, a M.Sc. in Computer Science from Katholieke Universiteir Leuven, Belgium, and a Ph.D. in Computer Science from McGill University, Canada. Kostas is working in the areas of software analysis, service computing, and model driven engineering.