TOWARDS THE RETRIEVAL OF REUSABLE SOFTWARE ARTIFACTS

BY

HAMZA ONORUOIZA SALAMI

A Dissertation Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

In

COMPUTER SCIENCE AND ENGINEERING

May 2014

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN- 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **Hamza Onoruoiza Salami** under the direction of his thesis advisor and approved by his thesis committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE AND ENGINEERING.**

Dr. Moataz Ahmed
(Advisor)

Dr. Umar Al-Turki
Department Chairman

Dr. Mohammad Alshayeb
(Member)

Dr. Salam A. Zummo
Dean of Graduate Studies

Dr. Radwan Abdel-Aal
(Member)

12/16/14
Date

Dr. Farag Azzedin
(Member)

Dr. Nasir Al-Darwish
(Member)

Dedicated to my parents

Acknowledgments

In the name of Allah, the Most Gracious, the Most Merciful

First and foremost, all praise is due to Allah for his countless blessings in my life.

I acknowledge the support of King Fahd University of Petroleum and Minerals throughout this research.

I am greatly indebted to my advisor, Dr. Moataz Ahmed, for his encouragement, support, and constructive advice. I have learned not only from his rich knowledge and experience in software engineering but also from his remarkable personality. I wish to thank my dissertation committee members, Dr. Mohammad Alshayeb, Dr. Radwan Abdel-Aal, Dr. Nasir Al-Darwish and Dr. Farag Azzedin, for their help, support, and contributions. Members of the Integrated Reuse Environment (IRE) project really enhanced my understanding of software engineering.

The Nigerian Community helped in no small way, to make KFUPM a home away from home.

Words cannot express my gratitude to my parents for showing me the value of education and helping me to acquire it.

I would not be writing these lines if not for the patience and understanding of my wife and two gems; Salmaan and Fatima. I plan to make up for all the loneliness my absence caused you all.

Contents

List of Tables

List of Figures

LIST OF ABBREVIATIONS

CBR	:	Case Based Reasoning
CSA	:	Cuckoo Search Algorithm
GA	:	Genetic Algorithm
HC	:	Hill Climbing
IR	:	Information Retrieval
LCS	:	Longest Common Subsequence
LCSMM	:	Longest Common Subsequence of Matching Messages
MAP	:	Mean Average Precision
MDE	:	Model Driven Engineering
MOOG	:	Message Object Order Graph
OWL	:	Ontology Web Language
PSO	:	Particle Swarm Optimization
RQ	:	Research Question
SA	:	Simulated Annealing
SLOC	:	Source Lines of Code
SME	:	Structure Mapping Engine

TE	:	Textual Entailment
TS	:	Tabu Search
UML	:	Unified Modelling Language
XMI	:	XML Metadata Interchange

ABSTRACT

Full Name : Hamza Onoruoiza Salami

Thesis Title : Towards the retrieval of reusable software artifacts

Major Field : Computer Science and Engineering

Date of Degree : May 2014

The benefits of software reuse include accelerated development, reduced cost, reduced risk and effective use of specialists. Early-stage reuse maximizes these benefits, because it allows subsequent reuse of later stage artifacts derived from earlier artifacts. Software is typically modeled from different viewpoints such as structural view, behavioral view and functional view. Unified Modeling Language (UML) is the *de facto* modeling language used by software developers during the initial stages of software development such as requirements engineering, architectural and detailed design. In this dissertation, we reviewed existing UML reuse works and classified them as multi-view or non-multi-view, based on their retrieval approaches. Because early-stage multi-view artifacts often consist of a set of models, we identified a number of important issues regarding mapping of entities during multi-view retrieval of UML models. In response to the raised issues, we have described a system for reusing UML artifacts. Within the reuse system, a pre-filtering stage helps to select a subset of repository models which will be considered during the retrieval stage. A retrieval stage assesses the similarity of query and shortlisted repository artifacts, and ranks them. Similarity assessment comprises matching and similarity scoring. Matching establishes a one-to-one mapping between similar entities in two models, while similarity scoring returns a similarity value between the models based

on the mapped entities. Due to the computational complexity involved in exhaustively matching entities in sets of models to be compared, heuristic search techniques are used for entity matching. Our techniques resulted in a Mean Average Precision of up to 98.50%, and the correlation between similarity scores and estimated reuse effort reached 0.84.

ملخص الرسالة

الأســـــم : حمزه أونورويزا سلامي

عنوان الرسالة : نحو إسترجاع أجزاء البرمجيات التي يمكن إعادة إستخدامها

التخصص الرئيسي : علوم وهندسة الحاسب الآلي

تاريخ التخرج : مايو 2014

هنالك العديد من الفوائد لإعادة استخدام البرمجيات ومنها التنمية المتسارعة، وانخفاض التكلفة، وانخفاض المخاطر والاستخدام الفعال للمتخصصين. لا سيما أن إعادة الإستخدام في مرحلة مبكرة من عملية تطوير البروجيات يزيد هذه الفوائد، لأنه يسمح بإعادة استخدام الإجزاء اللاحقة من البرمجيات بناء على إستخدام الاجزاء السابقة والتي بنيت عليها الاجزاء اللاحقة. وعادة ما يتم بناء البرمجيات من وجهات نظر مختلفة مثل عرض الهيكلية، وعرض السلوكية وعرض الوظيفة. لغة النمذجة الموحدة (UML) هو في الواقع لغة النمذجة المستخدمة من قبل مطوري البرمجيات خلال المراحل الأولى من تطوير البرمجيات مثل المتطلبات الهندسية والمعمارية والتصميم التفصيلي. في هذه الأطروحة، استعرضنا الاعمال السابقة المتاحة والمتعلقة بإعادة استخدام UML وقد قمنا بتصنيفها إلى صنفين رئيسين ذات وجهات النظر المتعددة أو ذات وجهة النظر وذلك إعتمادا على المنهجية المستخدمة لاسترجاع البرمجيات. الجدير بالذكر ان اجزاء البرمجيات ذات وجهات النظر المتعددة في المراحل المبكرة غالبا ما تتكون من مجموعة من النماذج، لذلك فقد قمنا بتحديد عددا من القضايا المهمة المتعلقة بربط كيانات نماذج UML المختلفة عند استرجاعها مع اعتماد وجهات النظر المتعددة. بناءا على المسائل التي تم مناقشتها، فقد قمنا بتوصيف نظام لإعادة استخدام اجزاء نماذج UML. بالاصافة الى ذلك ، ضمن منظومة إعادة الاستخدام، قمنا بعمل مرحلة ما قبل الترشيح والتي تساعد على تحديد مجموعة فرعية من نماذج المخزون والتي سيتم النظر فيها خلال مرحلة استرجاعها. في مرحلة الااسترجاع يتم تقييم و ترتيب التشابه بين الاستعلام والاجزاء المختصرة الموجوده في مخزن النماذج. ويتألف تقييم التشابه من جزئين رئيسين هما المطابقة و سجل التشابه. في المطابقة يتم ربط واحد الى واحد بين الكيانات المماثلة في نموذجين، في حين يقوم سجل التشابه بإرجاع قيمة التشابه بين النماذج القائمة على الكيانات المعينة. وبسبب ذلك التعقيد الحسابي الموجود في مطابقة كيانات موجوده في مجموعات من النماذج المراد مقارنتها،

فإنه توجب علينا استخدام تقنيات البحث الارشادي لمطابقة الكيانات. بإستخدام تقنياتنا المقترحة فقد حصلنا على نتائج تصل في متوسط الدقة إلى 98.50٪، وبلغ الارتباط بين درجات التشابه والجهد المقدر لإعادة استخدامها 0.84.

Chapter 1

INTRODUCTION

In this chapter, we briefly describe software reuse and our motivation for undertaking this study. Furthermore, we list our technical contributions and describe the organization of the entire dissertation.

1.1 Software Reuse

Software reuse is the creation of software using previously developed software rather than from scratch [1]. It helps to prevent or minimize *'reinventing the wheel'* during software development. The benefits of software reuse include accelerated development, reduced overall cost, increased dependability, effective use of specialists and reduced risk [2]. However, these benefits are not without any downside. Some of the challenges of software reuse are increased effort to create and maintain component libraries, effort to find and adapt reusable components, lack of tool support, the '*not-invented-here'* syndrome and increased maintenance cost [2, 3].

There are two types of software reuse: systematic reuse and opportunistic reuse [3].

- Systematic reuse: During systematic or deliberate reuse, purposefully constructed software components are utilized during the development of new software. This results in robust, well documented and thoroughly tested artifacts. However, it

requires time, effort and resources which some organizations are unwilling to sacrifice because there are no guarantees that such components will be reused in the future [3].

- Opportunistic reuse: In opportunistic or accidental reuse, software developers realize that previously developed components can be used in new software products. This type of reuse is simpler, but components may not be in the best form to be reused.

The retrieval techniques reported in this dissertation can be utilized during either systematic or opportunistic reuse.

Both types of software reuse can be carried out in four phases. These phases are representation, retrieval, adaptation and incorporation [4]. At the representation stage, a query (i.e., a model of the new software) is presented. During retrieval, a software component which is similar to the query, and whose adaptation cost is minimal is selected from the components library (repository). The retrieved component is modified to suit the needs of the new system during the adaptation phase. Finally, the new component is stored in the repository so that it can be reused in the future.

1.2 General Problem Statement and Motivation

Different types of artifacts may be reused during software development. These artifacts include domain models, requirement specifications, design, documentation, test data and source code. The first three types of artifacts listed above are referred to as *early-stage* artifacts while the other types are referred to as *later-stage* artifacts [5]. The benefits of

reuse can be maximized if early-stage artifacts are reused, because this leads to reuse of corresponding later-stage artifacts [6].

Typically, early-stage artifacts such as requirement specifications and design artifacts are described using sets of models (such as Unified Modeling Language (UML) diagrams). These models commonly describe software systems from different perspectives. For example, a structural perspective may show the static relationship between various system elements, while a functional perspective may describe a system based on its functionality.

The general problem is that reusability assessment of existing software artifacts in a new situation needs to take into account the collective information contained in the multiple viewpoints representation of software systems. In other words, similarity of software systems should be evaluated in a consistent manner by simultaneously considering the different perspectives of the systems, rather than by simply aggregating similarity values obtained by independently considering the individual viewpoints. This work addresses the problem by investigating previous work on multi-view reusability assessment and providing solutions to overcome weaknesses identified in existing work.

In view of the benefits of early-stage artifacts reuse in particular, and software reuse in general, the objectives of this work can be summarized as follows:

- Efficiently retrieve software artifacts modeled from multiple perspectives, without introducing any inconsistencies
- Provide a method for systematically comparing sets of early-stage models

1.3 Contributions

This section summarizes the main contributions of this dissertation by describing techniques we have introduced and detailing our published and submitted works.

1.3.1 Survey of UML Reuse Works

At the beginning of the research, we carried out a comprehensive survey of existing works that have compared UML models for the purpose of software reuse. The survey studied the retrieval techniques, UML diagrams supported, tools provided and experiments carried out in various UML-based reuse works. One conference paper resulted from this work:

> **H. O. Salami**, and M. Ahmed (2013), "UML Artifacts Reuse: State of the Art", International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 3, No. 3, 2013, presented at the International Conference on Soft Computing and Software Engineering 2013 (SCSE'13), March 1-2, 2013, San Francisco, California, USA.

1.3.2 Techniques for Structural Similarity Assessment

Class diagrams are arguably the most popular of all the UML diagrams. They are commonly used to depict the structure of a system by showing the classes that make up the systems and the relationships between them. Class diagram similarity assessment was treated as a graph matching problem which involved entity matching and similarity scoring. Some of the contributions of this dissertation in the area of class diagram-based retrieval of software include: development of a similarity measure; identification of suitable features for computing the similarity of classifiers in a diagram; and

determination of a suitable approach for matching classifiers in a class diagram. Our retrieval technique resulted in the publication of two conference papers:

1. **H. O. Salami**, and M. Ahmed (2012), "A Framework for Class Diagram Retrieval using Genetic Algorithm", The 24th International Conference on Software Engineering and Knowledge Engineering (SEKE'12), July 1 – 2, 2012, Redwood City, San Francisco Bay, USA.
2. **H. O. Salami**, and M. Ahmed (2013), "Class Diagram Retrieval Using Genetic Algorithm", 12th International Conference on Machine Learning and Applications Miami, Florida, 2013, presented at the IEEE/ICMLA 2013, Dec 4 – 7, 2013, Miami, Florida, USA.

1.3.3 Techniques for Functional Similarity Assessment

Early in the development lifecycle, functionality of software systems is manifested in the sequence diagrams which realize the different use cases of the system. Thus, sequence diagram-based retrieval of software can be used to exploit the gains of early-stage reuse. Our work on functional similarity assessment could be split in three parts.

The first part computed the similarity of two sequence diagrams by comparing their graph representations using similarity measures and heuristic search techniques. In the second part, the good results obtained in the first part were improved when we formulated similarity assessment of two sequence diagrams as a search for the longest common subsequence of their matching messages. In practice, several sequence diagrams are used to model a software system hence in the last part of our work on functional similarity assessment we extended the techniques developed in the second part to cater for the comparison of two sets of sequence diagrams. The following paper resulted from our work on functional similarity assessment:

H. O. Salami and M. A. Ahmed, "Retrieving Sequence Diagrams Using Genetic Algorithm," *to appear in the* Proceedings of The 11th International Joint Conference on Computer Sciences and Software Engineering, May 14 – 16, 2014, Chonburi, Thailand.

1.3.4 Techniques for Behavioral Similarity Assessment

UML state machine diagrams model the behavior of individual objects in a system by showing how different events cause the object to change its state. In this part of the dissertation, we proposed a method of representing state machine diagrams as directed graphs. Furthermore, the dissertation described a similarity measure as well as a matching technique for comparing the graph representation of state machine diagrams.

1.3.5 Techniques for Multi-view Similarity Assessment

Software is typically modeled from different viewpoints rather than from a single perspective. This part of the dissertation studied existing UML-based reuse works and identified those that utilized multi-view retrieval techniques. In addition, a number of important issues regarding mapping of entities during multi-view retrieval of UML models were raised. Furthermore, three architectures for multi-view retrieval were presented, and the ways they tackled the identified issues were discussed. The following papers resulted from our work in this part:

1. **H. O. Salami**, and M. Ahmed (2013), "A framework for reuse of multi-view UML artifacts", International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 3, No. 3, 2013, presented at the International Conference on Soft Computing and Software Engineering 2013 (SCSE'13), March 1 – 2, 2013, San Francisco, California, USA.

2. M. A. Ahmed and **H. O. Salami**, "On multi-view based retrieval of software," *under preparation*

1.3.6 Technique for Pre-filtering

When the repository contains many models, retrieval time may be degraded to an unacceptable point. This part of the dissertation describes a fast way of identifying a subset of repository projects that are potentially similar to a query model. These shortlisted projects are then compared in a more computationally demanding retrieval stage. A number of suitable metrics that provide size and complexity information about projects was identified for computing the similarity of query and repository projects during the pre-filtering stage.

1.4 Organization

The rest of this dissertation is organized as follows: Chapter 2 presents background information on UML and discusses some important issues to be considered during multi-view retrieval. Chapter 3 reviews related work on software artifacts reuse. Chapter 4 describes our research questions and objectives as well as our approach for comparing query and repository models. In addition, the concept of operation of our reuse system is discussed in the chapter. Chapter 5, Chapter 6, Chapter 7 and Chapter 8 are concerned with structural, functional, behavioral and multi-view similarity assessment, respectively. In Chapter 9, a technique for filtering repository models prior to full-fledged similarity assessment is described. Experimental results are presented in Chapter 10. Finally, we conclude the dissertation and provide directions for future work in Chapter 11.

Chapter 2

BACKGROUND

In this chapter, we provide a background on UML and raise several important issues regarding retrieval of multi-view UML artifacts.

2.1 Unified Modeling Language (UML)

During the early stages of software development, requirement specifications are typically modeled from related but different viewpoints [7]. The division into different views is arbitrary, and often includes at least three views namely structural, functional and behavioral views [8-10]. Each view represents one aspect of the system to be developed; collectively, they provide a complete specification of the system. One or more types of diagrams provide a visual notation for concepts in each view [7].

UML is a general-purpose modeling language maintained by the Object Management Group (OMG), a consortium of companies. It provides diagrams for visualizing, specifying, constructing and documenting the artifacts of a software-intensive system [11]. UML is widely used to model early-stage object oriented artifacts such as analysis and design models [12]. The UML taxonomy of diagrams partitions the various diagrams into two categories: structure diagrams and behavior diagrams [13]. Structure diagrams such as class, component and object diagrams document the static structure of system

objects. On the other hand, behavior diagrams like state machine, activity and timing diagrams show the dynamic behavior of system objects.

The UML taxonomy of diagrams considers only structure and behavior diagrams, without any category for the functional aspect of software systems [13]. However, because use cases define the functionality of a system [14], use case diagrams can be considered as representing the functional view. Moreover, one or more sequence diagrams is typically used to realize each use case so sequence diagrams can also be considered as belonging to the functional view. The most recently released UML specification, UML 2.4.1, describes fourteen diagrams as shown in Figure 2.1. In the sequel, we describe class diagrams, sequence diagrams and state machine diagrams which are representative of structural, functional and behavioral views of software systems [7].

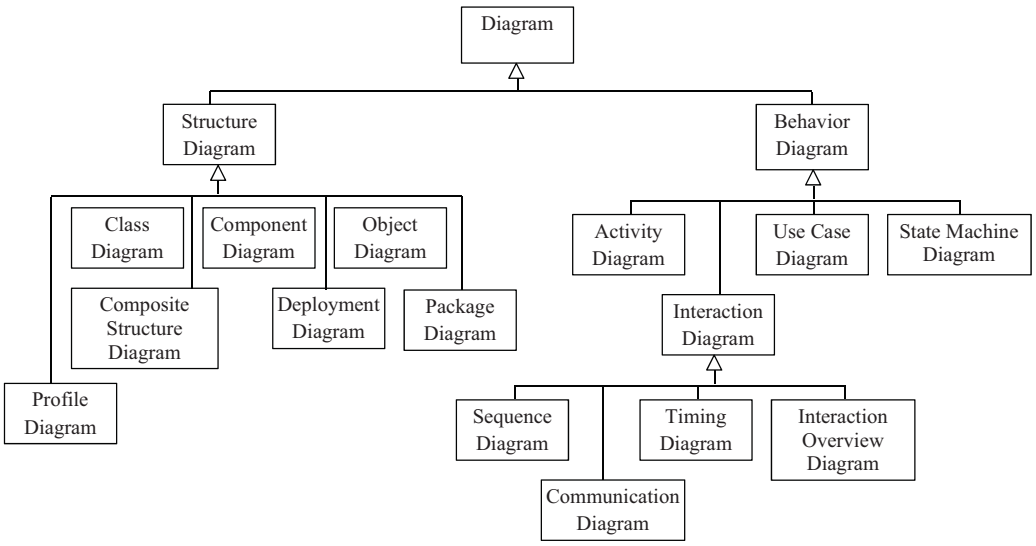


Figure 2.1: Taxonomy of UML diagrams as a class diagram [13].

2.1.1 Class Diagram

A class is a set of objects that share the same attributes and methods. A class diagram depicts the structure of a system by showing the system's classes and the relationships among the classes. Attributes represent class properties while methods are actions that the class can perform. Figure 2.2 shows a class diagram for an ATM (Automated Teller Machine) system.

2.1.2 Sequence Diagram

A sequence diagram shows the interactions between objects arranged in time order. It depicts the functionality of use case scenarios by showing objects and messages that are passed between these objects in a use case. The vertical dimension in a sequence diagram represents time, while the horizontal dimension represents the objects participating in an interaction [15]. A sequence diagram for *withdraw money* scenario is shown in Figure 2.3.

2.1.3 State Machine Diagram

State machine diagrams are used to model the behavior of individual system entities such as objects [13]. They show how an object responds to events according to its current state, and how it enters new states [16]. The basic notational elements of a state machine diagram are a rounded rectangle (state), an arrow (transition), a filled circle (initial state) and a hollow circle containing a smaller filled circle (final state). A state machine diagram for objects of the *ATM Transaction* class of Figure 2.2 is shown in Figure 2.4.

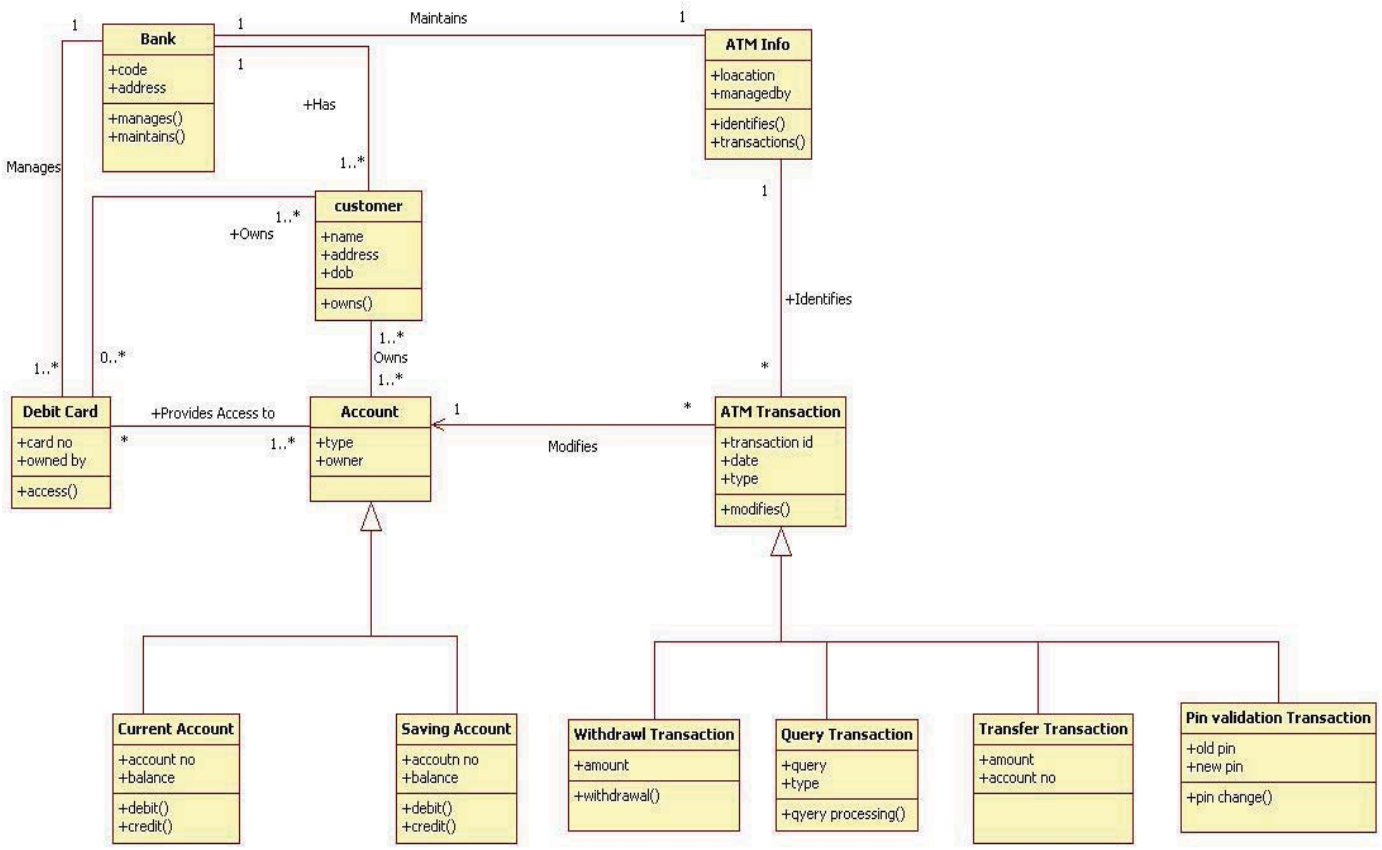


Figure 2.2: Class diagram for ATM system

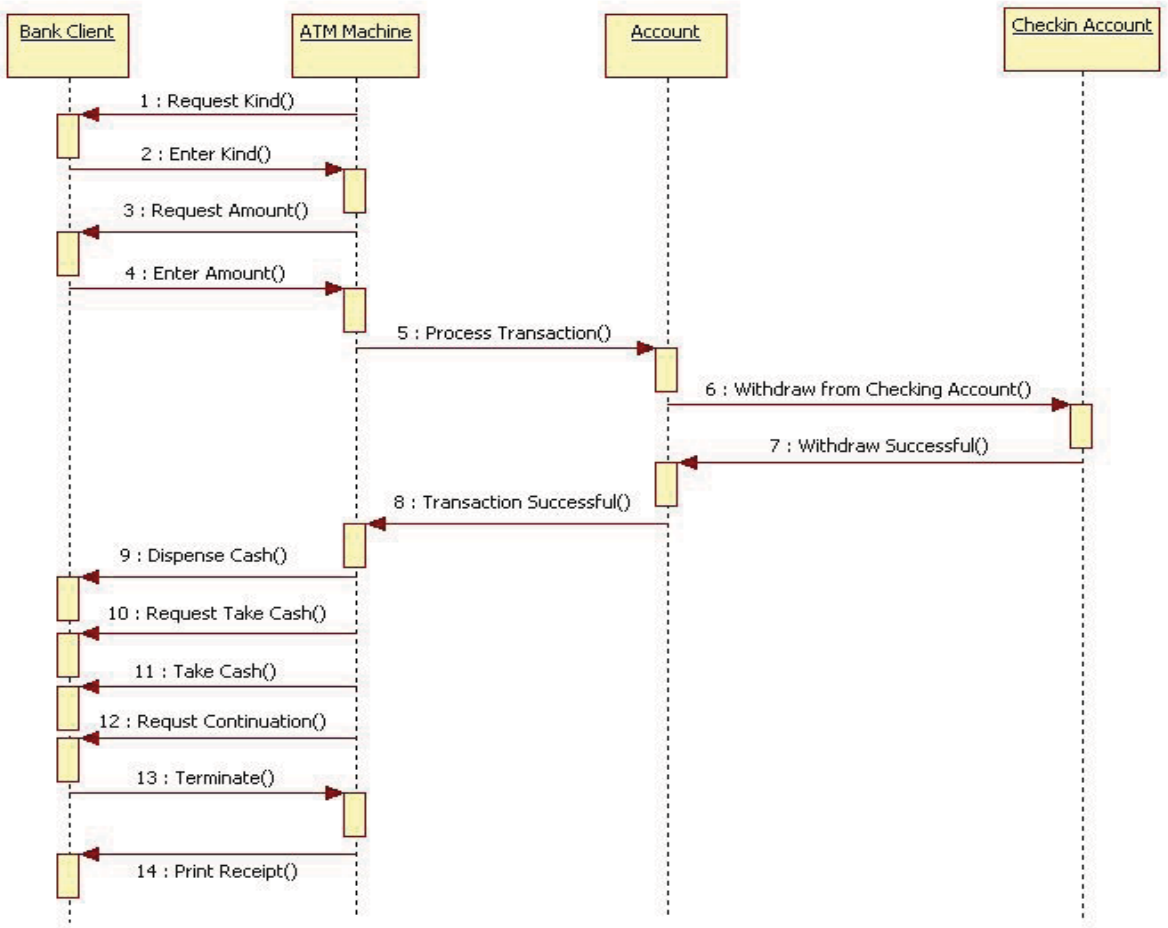


Figure 2.3: Sequence diagram for *withdraw money* use case

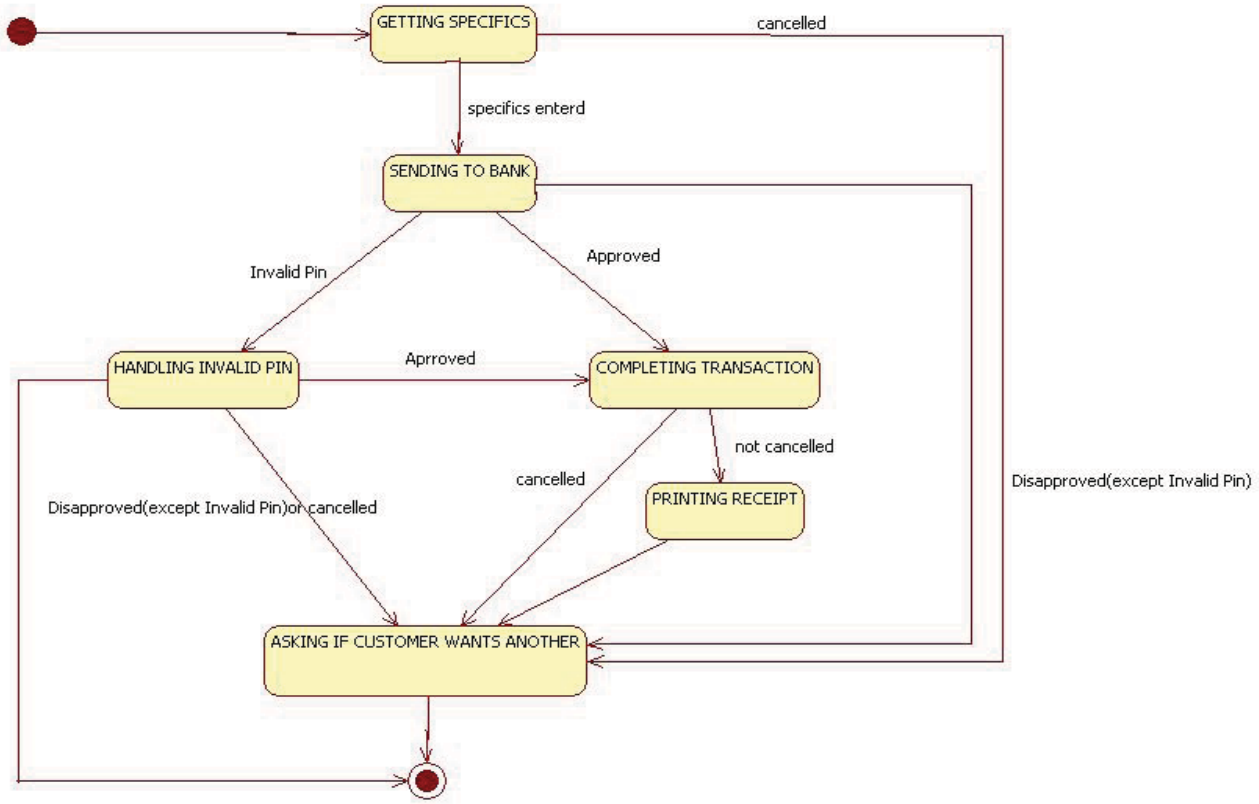


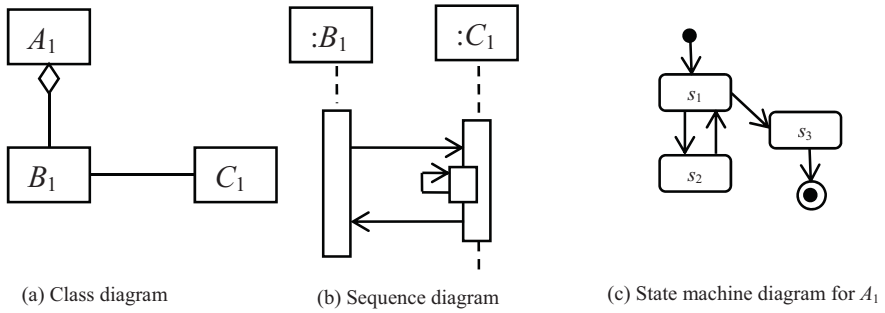
Figure 2.4: State machine diagram for *ATM Transaction*

2.2 Important Issues in Multi-view Retrieval

In this section, we identify three important issues regarding mapping of model entities that should be taken into consideration during comparison of multi-view requirement specifications.

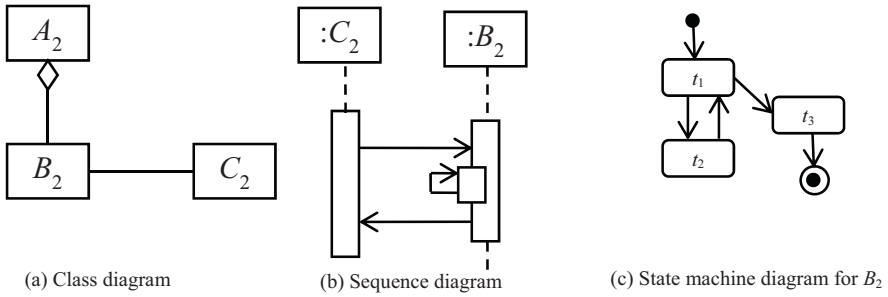
2.2.1 Consistent Mapping of Classes in Class Diagrams and Sequence Diagrams

We provide an illustrative example to underscore this issue. Assume a query requirement specification Q_1 is to be compared with a requirement specification R_1 from the repository. Both Q_1 and R_1 have one class diagram, one sequence diagram and one state machine diagram as shown in Figure 2.5 and Figure 2.6, respectively. While comparing requirement specifications Q_1 and R_1, a retrieval technique which merely computes multi-view similarity as an aggregation of similarity values from individual views would produce a wrong overall similarity value. Similarly, if the diagrams are compared in stages, maximum similarity value is obtained for both class diagrams in one stage, and maximum similarity value is obtained for the sequence diagrams in the other stage. Both approaches produce inaccurate similarity scores because of the inconsistent mapping of classes in the class diagrams and sequence diagrams ($A_1 \rightarrow A_2$, $B_1 \rightarrow B_2$, $C_1 \rightarrow C_2$ in the class diagrams, and $B_1 \rightarrow C_2$, $C_1 \rightarrow B_2$ in the sequence diagrams).



(a) Class diagram (b) Sequence diagram (c) State machine diagram for A_1

Figure 2.5: Requirement specification Q_1 containing three diagrams



(a) Class diagram (b) Sequence diagram (c) State machine diagram for B_2

Figure 2.6: Requirement specification R_1 containing three diagrams

2.2.2 Consistent Mapping of Classes in Class Diagrams and State Machine Diagrams

State machine diagrams are used to model the behavior of system elements such as objects (that is, class instances) [13]. They show how an object responds to events according to its current state, and how it enters into new states [16]. Just as classes should be consistently mapped in class and sequence diagrams, it is important to ensure that classes are consistently mapped when comparing two models containing class and state machine diagrams.

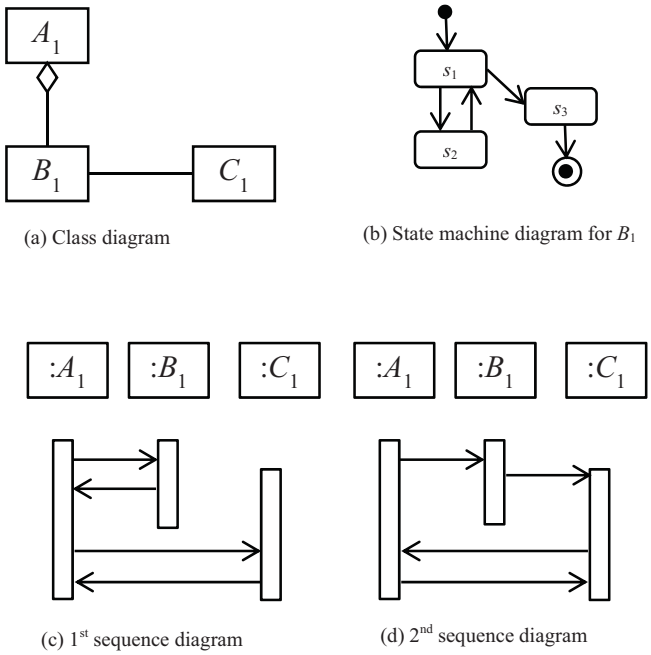
Consider the state machine diagrams in Figure 2.5 (c) and Figure 2.6 (c). When the two identical state machine diagrams are compared to each other without considering other diagrams, their similarity score is maximal. However, the multi-view similarity score between Q_1 and R_1 is not expected to be maximal for the following reason. The state machine diagrams of Figure 2.5 (c) and Figure 2.6 (c) depict the behavior of classes A_1 and B_2, respectively. Comparing the two state machine diagrams means that A_1 is mapped to B_2. However, it can be observed from the class diagrams of Figure 2.5 (a) and Figure 2.6 (a) that A_1 is mapped to A_2, and not B_2.

2.2.3 Systematic Mapping of Multiple Sequence Diagrams in two Requirements

During the requirements phase of a software project, use cases are used to specify the functionality of a system. One or more sequence diagrams is then used to realize each use case [7]. Thus, it is common for requirement specifications to contain several sequence diagrams. An important issue is how to efficiently compare the sets of sequence diagrams in two requirement specifications.

The requirement specifications in Figure 2.7 and Figure 2.8 each have two sequence diagrams. There are two possible ways of exhaustively mapping these sequence diagrams in order to compare them: Figure 2.7 (c) $\rightarrow$ Figure 2.8 (c) and Figure 2.7 (d) $\rightarrow$ Figure 2.8 (d); and Figure 2.7 (c) $\rightarrow$ Figure 2.8 (d) and Figure 2.7 (d) $\rightarrow$ Figure 2.8 (c). However, when requirement specifications contain many sequence diagrams, exhaustively matching them becomes computationally exhaustive. For example, if two requirement specifications have 9 sequence diagrams and 10 sequence diagrams

respectively, there are 10!/(10-9)! = 3,628,800 different ways of exhaustively matching the sequence diagrams in order to compare them.



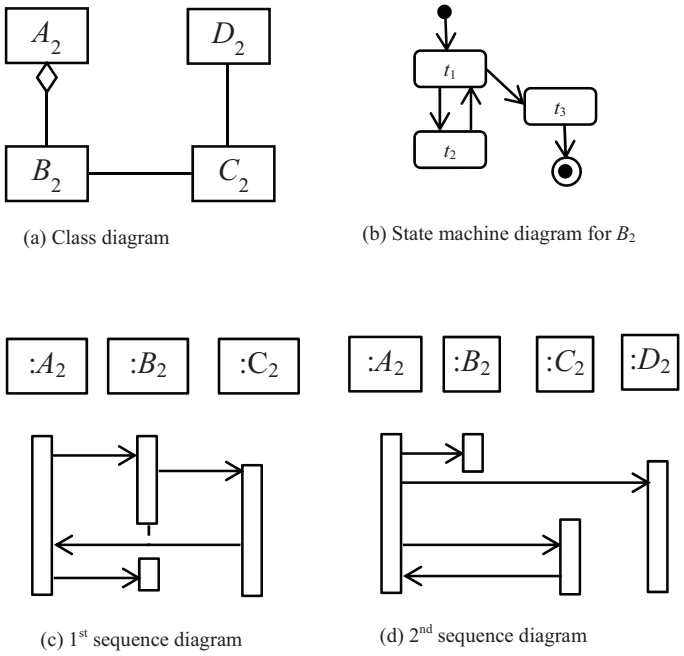
(a) Class diagram

(b) State machine diagram for B_1

(c) 1st sequence diagram

(d) 2nd sequence diagram

Figure 2.7: Requirement specification Q_2 containing four diagrams



(a) Class diagram

(b) State machine diagram for B_2

(c) 1st sequence diagram

(d) 2nd sequence diagram

Figure 2.8: Requirement specification R_2 containing four diagrams

Chapter 3

Literature Review

In this chapter, we present existing works on UML artifacts reuse in chronological order. We also identify existing reuse works that utilize multi-view retrieval techniques. We consider a reuse work as multi-view, only if it compares artifacts that have at least one structure diagram and one behavior diagram from the UML taxonomy of diagrams. Moreover, we describe which of the existing studies has tackled the issues raised in Section 2.2. The three issues are: consistent mapping of classes in class diagrams and sequence diagrams (I_1); consistent mapping of classes in class diagrams and state machine diagrams (I_2); and systematic mapping of sequence diagrams in two sets of sequence diagrams (I_3). Section 3.1 describes existing UML-based reuse works, while Section 3.2 discusses the limitations of these works.

3.1 Existing UML-based Reuse Works

Early work on UML artifacts reuse focused on comparing use case descriptions contained in software artifacts. Not surprisingly, these studies have used information retrieval (IR) techniques. IR involves locating documents (usually text) satisfying an information need from a large collection of documents [17]. IR techniques are applicable to software artifacts that contain considerable amount of text (for example, use case descriptions), but do not take into consideration the structural information contained in artifacts [18].

Authors in [19] computed the similarity between requirement specifications of query and repository models using IR techniques. The requirements were in the form of use case flow of events. Events occurring in a particular domain were grouped into clusters based on the lexical meaning of words describing them. Each use case's flow of events was represented using the vector space model (i.e., as a multi-dimensional vector). Each dimension represented the number of events belonging to a particular cluster. Finally, the cosine distance measure was used to determine the degree of similarity between the two requirement specifications.

In [20], IR techniques were used for scenario management and reuse. Each scenario was represented as a set of attributes: goals; authors; events; actors; actions; and episodes (i.e., named subsequence of events in a scenario). The similarity between two scenarios was computed as the degree of overlap between their set of attributes using the Dice similarity coefficient.

Ali and Du [21] have retrieved repository models in two steps: classification and retrieval. During classification a model is described from six perspectives/facets which capture the model's functional requirements. In the retrieval step, a *'discrepancy ratio'* similarity metric is computed using the degree of commonality in the descriptor terms of query and repository models. Another similarity metric used during retrieval was *'conceptual closeness'*, computed from different facets, each of which is represented by terms forming a taxonomy. Because software models containing class, object, activity, state machine and collaboration diagrams can be retrieved for reuse, we consider their work as utilizing a multi-view reuse approach.

Gomes et al. [22-26] have used Case Based Reasoning (CBR) and the WordNet lexical ontology to retrieve software designs. CBR is a problem solving paradigm in which new problems are solved by reusing the solution to similar past problems. Past designs were stored as class diagrams (cases) in a repository (case base). Three types of objects can be retrieved: classes, interfaces and packages. Retrieval was carried out in two phases: a computationally inexpensive phase in which a fixed number of relevant cases are chosen by using WordNet relations to index the cases; and a computationally demanding ranking phase when previously chosen objects are ordered using similarity metrics. After retrieval, one or more retrieved cases is automatically adapted to build a new case.

In [5] class diagrams were retrieved using a set of similarity metrics. The metrics were computed based on semantic relatedness of class names, class attribute names, and class method names, as well as the class fingerprints (i.e., the set of sub classes, super classes and interfaces implemented by a class). Furthermore, the Hungarian Algorithm was used to ensure one-to-one mapping of classes in two class diagrams.

Robinson and Woo [27] applied graph matching in the retrieval of sequence diagrams. Both query and repository sequence diagrams were represented as conceptual graphs in which UML metamodel elements are encoded as vertices and UML metamodel associations are encoded as edges. A graph matching algorithm (SUBDUE [28]) was then applied to find the similarity between both conceptual graphs. Given a query sequence diagram, the algorithm finds similar substructures in repository sequence diagrams. Users could supply two parameters for limiting the search for common substructures in sequence diagrams: *beam*, to restrict the search by breadth; and *limit*, to restrict the search

by number of expansions. The authors indicated that the matching technique can also be applied to use case diagrams and class diagrams.

Authors in [29] described a framework for retrieving UML artifacts from a repository in two stages: indexing and retrieval. During indexing a UML model in XML Metadata Interchange (XMI) form is stored in a relational database system. During retrieval, query and repository models are compared using either query inclusion or query similarity. The query inclusion technique searches for repository models that subsume the query model by formulating a SELECT statement and querying the database of repository models. On the other hand, query similarity is composed of topological and semantic similarity. Topological similarity is the Euclidean distance between two vectors where each vector dimension represents a different type of relationship and specifies the number of such relationships in a diagram. Semantic similarity is computed from the degree of overlap of terms occurring in both models and the distance (within a thesaurus) of terms/concepts occurring in relationships.

In [30] similar class diagrams were retrieved from a repository in three stages using the class names and weights. Each class is assigned a weight reflecting its degree of influence in the class diagram by considering the class's relationship types, and the navigability and multiplicity of its association ends. In the first stage which is a filtering stage, a fixed number of repository diagrams are selected based on the number of common class names in query and repository diagrams. In the following stage, a subset of previously selected class diagrams are chosen by considering the class weights in query and selected repository diagrams. Finally, the chosen repository diagrams are ranked by converting them to weighted graphs and applying a shortest path algorithm.

Similarity between sets of sequence diagrams is computed using two nested levels of genetic algorithm (GA) in [31]. At the lower level, similarity is measured by mapping the classes in two sequence diagrams (using GA), and considering the number of matching and differing method calls. At the higher level, GA was used to map sequence diagrams in one model to sequence diagrams in the other model. We note that even though the work correctly handles systematic mapping of sequence diagrams (I_3), it is not regarded as a multi-view reuse work because none of the UML structure diagrams are considered during retrieval.

In [32] query and repository UML models are transformed from their XMI representations to specifications written in first order logic. The specifications are then matched, guided by some meta-knowledge (set of rules). Their approach supports matching of class, sequence, use case and communication diagrams.

In the ReDSeeDS project [18, 33-35], artifacts from previously developed software are stored as cases comprising a problem (i.e., requirements), and a solution (i.e., architecture, design and implementation code). In essence, requirements act as case indexes that have links to corresponding artifacts. Requirements are represented in a Requirements Specification Language (RSL) in three possible formats: scenarios written in less formal natural language sentences; scenarios written in more formal constrained Structured English sentences; and using UML activity and sequence diagrams. During retrieval, requirements for a new project are compared with requirements of all projects in the case base. The most similar case is returned for reuse, based on the intuition that systems with similar requirements should have many artifacts in common. Computation of similarity scores for two requirements relies on the WordNet lexicon and either IR

techniques (for natural language sentences) or graph matching techniques (for constrained language scenarios). During adaptation, a transformation engine generates new interaction diagrams as well as application and business logic code for the new requirement from the retrieved case. None of the UML structure diagrams are considered during retrieval, hence, we do not consider their work as a multi-view reuse work.

Kotb [36] describes an approach for retrieving similar use case descriptions using Textual Entailment (TE), a natural language processing technique. A text *T* entails another text *H* if the meaning of *H* can be inferred from that of *T*. With the aid of the WordNet lexicon, the author proposes comparing the summarized descriptions of query and repository use cases. Any repository use case whose summarized flow of events is entailed by that of the query is retrieved for reuse. During adaptation, a new scenario is generated from the entailed repository scenario using WordNet.

Park and Bae [37] adopted a two-stage multi-view approach for retrieving repository artifacts. In the first stage, they determined the similarity score of two class diagrams using the Structure Mapping Engine (SME). The SME software works based on the structure mapping theory -an analogical mapping technique- which allows knowledge to be mapped in two domains by considering relational commonalities of objects in the domains regardless of the objects involved in the relationships. Based on the filtering, a subset of repository UML models are selected. During the second stage, sequence diagrams in the shortlisted models are converted to Message-Object-Order-Graphs which are then compared using a graph matching algorithm.

In [38] information about actors and use cases contained in use case diagrams, as well as additional information entered by users are integrated into an Ontology Web Language (OWL) base ontology. The ontology is stored in a repository using a relational database system. In order to reuse past use case diagrams, the (re)user enters query parameters such as actor, use case and project names. The authors' tool queries the OWL ontology and returns a list of relevant use case diagrams.

Two types of domain-specific ontologies have been used for class diagram retrieval in [39]. 'Application ontologies' were developed to measure semantic similarity of UML class diagram classifiers as well as relationships. A 'domain ontology' was then used to measure the semantic similarity between classifiers names. Overall similarity between class diagrams was computed as a weighted sum of both similarity scores. The 'application ontologies' are built once, while the 'domain ontology' is built for each new problem domain.

In [40, 41] we compute the structural similarity of two class diagrams using an inexact graph matching technique. With the aid of a lookup table containing difference values for various class diagram relationships, the similarity score was computed from the adjacency matrix representation of each class diagram. The adjacency matrix is obtained by considering classes as nodes, and relationships between them as edges. GA was used to select optimal mapping of classes in both class diagrams. Assuncao and Vergilio [42] built on the retrieval technique we introduced in [40], even though they use Particle Swarm Optimization for matching rather than GA.

Table 3.1 summarizes the different works. Each work is identified by its first author and reference number. We observe that very few existing UML artifact reuse works can be considered as multi-view. Furthermore, it can be noticed that consistent and systematic mapping of artifact entities (see Section 2.2) has not caught the attention of researchers.

3.2 Limitations of Existing UML-based Reuse Works

The main limitations of previous UML reuse works are:

1. Ignoring structural information contained in UML models: Previous studies (e.g. [19-21]) that considered only the text contained in UML models do not take into account the structural content of UML models. As a result of this limitation, two structurally different models containing similar text may be regarded as identical.
2. Lack of information on how to handle multiple diagrams: It is not uncommon that early-stage artifacts contain several UML diagrams of the same type (e.g. multiple sequence diagrams and/or multiple state machine diagrams). Few works have discussed how to avoid exhaustively comparing multiple diagrams of the same type. Blok and Cybulski [19] use a heuristic that avoids comparing all combinations of use case event flows. Ahmed [31] uses GA to find suitable mappings of sequence diagrams in two sets of sequence diagrams. On the other hand, many works (e.g. [27, 32, 36, 37]) either exhaustively compare diagrams in two sets of UML diagrams, or do not explicitly mention how to compare sets of UML diagrams.
3. Little research on UML state machine diagram-based retrieval: To the best of the author's knowledge, only the work of Ali and Du [21] considers state machine

diagrams during retrieval. However, as previously mentioned, their work utilizes the text contained in UML artifacts, but ignores the structural content.

4. Lack of consideration of consistency across views: As mentioned in Section 2.2, while comparing early-stage artifacts, consistency should be maintained across the different views. The existing UML works do not take this issue into account. For example, Park and Bae [37] compare class diagrams in one stage, and compare sequence diagrams in another stage. This manner of similarity assessment that compares UML diagrams in one view independent of diagrams in other views may lead to incorrect similarity scores as described in Section 2.2.1.

Table 3.1: Summary of various UML artifacts reuse works

Author	Retrieval Technique(s)	UML Structure Diagrams Supported		UML Behavior Diagrams Supported						Multi -view	Handling of Consistency Issues		
		CD	OD	SD	UCD	AD	SMD	COD	CMD		I_1	I_2	I_3
Blok [19]	ON, IR				√								
Alspaugh [20]	IR				√								
Ali [21]	IR, ON	√	√	√		√	√	√		√			
Gomes [23]	CBR, ON	√											
Rufai [5]	ON, Hungarian algorithm	√											
Robinson [27]	Graph matching	√		√	√								
Llorens [29]	IR, Database querying, ON	√											
Channarukul [30]	Shortest path algorithm	√											
Ahmed [31]	GA			√									√
Khalifa [32]	First Order Logic matching	√		√	√				√				
Bildhauer [33]	IR, Graph Matching, CBR, ON			√	√	√							
Kotb [36]	Natural Language Processing				√								
Park [37]	Analogy, Graph Matching	√		√						√			
Bonilla-Morales [38]	ON, Database querying				√								
Robles [39]	ON	√											
Salami [40, 41]	Graph Matching, GA, ON	√											
Assuncao [42]	Graph Matching, PSO	√											
This dissertation	Graph Matching, GA, e.t.c.	√		√			√			√	√	√	√

CD = class diagram, OD = object diagram, SD = sequence diagram, UCD = use case diagram, AD = activity diagram, SMD = state machine diagram, COD = collaboration diagram, CMD = communication diagram, I_1 = consistent mapping of classes in class diagrams and sequence diagrams, I_2 = consistent mapping of classes in class diagrams and state machine diagrams, I_3 = systematic mapping of sequence diagrams in two sets of sequence diagrams, ON = WordNet or other ontology.

Chapter 4

Research Problem and Research Approach

This chapter begins by posing our research questions and stating our research objectives. Thereafter, we present an overview of our approach for retrieving software artifacts for reuse. Finally, a brief description of heuristic search techniques is provided.

4.1 Research Questions

Even though software reuse has traditionally been carried out at the source code level [43], significant research has also been done regarding the reuse of UML artifacts. Some of the motivations for shifting focus from code-centered reuse to the reuse of software artifacts modeled using UML include:

- UML is widely used by software developers during the initial stages of software development such as requirements engineering, architectural and detailed design.
- With the advent of Model Driven Engineering (MDE), models are considered as *first class artifacts* i.e., the main artifacts during software development [44].
- There is available research on *model-to-code* transformations [45-48].

Although substantial research has been carried out on reusing UML artifacts, the literature survey in Chapter 3 suggests that little research effort has been put in the

development of techniques for reusing software artifacts described from multiple viewpoints.

In this dissertation, we investigate the following key questions regarding reuse of multi-view UML artifacts (i.e. UML artifacts represented from multiple view points):

1. What measures are suitable for determining similarity of multi-view UML artifacts?
2. What are the appropriate matching techniques that can be employed during retrieval?
3. How does the matching technique affect the quality of retrieval?
4. How can we pre-filter models when the repository contains many artifacts?
5. What is the best framework for consistent multi-view similarity assessment?

Research Question (RQ) 1 is concerned with researching and developing similarity measures to compare model entities (such as classes and sequence diagrams) in the individual views, as well as an overall similarity measure for two requirement specifications. RQ 2 seeks to determine how search techniques (e.g. heuristic search techniques) can be used to map model entities. RQ 3 involves assessing the impact of the search technique on retrieval quality. For example, "Does matching with Particle Swarm Optimization (PSO) give better results compared with GA?" RQ 4 is concerned with researching and developing computationally inexpensive techniques for selecting an initial subset of requirement specifications from the repository that will be compared with the query requirement specification before a more computationally demanding retrieval stage. Finally, RQ 5 is concerned with investigating how higher-level (multi-view)

similarity values can be computed from lower-level (single-view) similarity values. Table 4.1 shows the sections of the dissertation that address each research question.

In order to answer the aforementioned research questions, we propose a multi-view retrieval method which utilizes information from three views of a UML model: structural view, behavioral view and functional view. Class diagrams, state machine diagrams and sequence diagrams would be considered as representative diagrams of the structural, behavioral and functional views, respectively.

We shall compare requirement specifications for new software to those of existing software systems contained in a repository. The corresponding artifacts (such as design, code and documentation) for the software system with the most similar requirements are returned for reuse, because it is expected that systems with similar requirements should have many other artifacts in common. Thus, it is assumed that requirement specifications for software systems in the repository contain class diagrams, state machine diagrams and sequence diagrams. The techniques we shall develop for comparing requirement specifications will also be applicable for comparing design stage artifacts modeled using UML.

4.2 Research Objectives

UML is the *de facto* modeling language for describing and designing software systems. Most of the existing studies on UML artifacts reuse have considered UML diagrams in the different views independently. To the best of our knowledge, the few existing works that can be considered as utilizing multi-view retrieval techniques do not consider consistency of model elements during retrieval. Consequently, the main goal of this

dissertation is to introduce consistent multi-view reuse techniques to maximize productivity and improve the quality of software products. In order to achieve these goals, the following objectives are defined:

- Develop efficient pre-filtering techniques to speed up artifacts retrieval from a large repository.
- Develop efficient similarity measures and matching techniques for model elements in functional, behavioral and structural views.
- Develop a consistent multi-view framework for effective ranking and retrieval of similar previous projects.

Table 4.1: Mapping of research questions to relevant sections of dissertation

Research Question (RQ)	**Section of dissertation where RQ was investigated**	**Section of dissertation where relevant experiments were carried out**
RQ1	5.1.2, 5.2, 6.1.2, 6.2.2, 6.3.1, 7.2, 7.5, 8.1, 8.2 and 8.3	Sections 10.3, 10.4, 10.5, 10.6, 10.7, 10.8
RQ2	5.1.4, 6.1.4, 6.2.3, 6.3.3 and 7.4	Sections 10.3, 10.4, 10.5, 10.6, 10.7, 10.8
RQ3	-	Sections 10.3 and 10.4
RQ4	Chapter 9	Section 10.9
RQ5	Chapter 8	Sections 10.6, 10.7 and 10.8

4.3 Research Approach

Our approach is hinged on the intuition that similar software systems have similar requirements. Thus, we shall compare requirement specifications of new projects (software systems) to be built with requirement specifications of existing projects stored

in a repository. Once the most similar requirements are found, the corresponding requirements can be adapted to meet the needs of the new software system. Moreover, since UML is the *de facto* language for modeling software requirements, we shall compare requirement specifications described using UML. In order to present a user with the most similar project from the repository, we utilize a two-stage technique that comprises pre-filtering and retrieval. These two stages are described next.

4.3.1 Pre-filtering

The aim of the pre-filtering stage is to minimize retrieval time by selecting a first set of repository artifacts, which will be assessed and ranked in the following stage. Pre-filtering is particularly important when the repository contains many projects. In this stage, metadata of the new requirement specification is compared with the metadata of each repository project. The metadata collected for each project include size and complexity metrics such as total number of classes in a class diagram, number of messages exchanged by objects in a sequence diagram, and the number of attributes and operations of classes. These metrics can be used to filter out repository projects whose sizes are significantly different from that of the new system.

In order to ensure that this stage is computationally inexpensive, the metadata are obtained from requirement specifications when entire projects are stored in the repository for the first time. They are also updated whenever changes are made to repository projects. However, the metadata for the new system are obtained in the pre-filtering stage, since the requirements of the new system become available to the reuse system only at this stage. Pre-filtering is covered in more detail in Chapter 9.

4.3.2 Retrieval

Retrieval is an important task during software reuse [42]. During retrieval, matching and similarity measures are employed to asses and rank the requirement specifications shortlisted in the pre-filtering stage. Matching refers to mapping an entity in one model (requirement specification) to another entity of the same type in the other model to be compared. Matching is therefore a combinatorial optimization problem. Once a pair of entities has been mapped, appropriate similarity measures can be used to compute a similarity score.

At the end of this stage, a ranked list of requirement specifications is presented to the reuser. Requirement specifications at the top of the list are most similar to the new requirement specifications, thus adaptation of the corresponding artifacts (for example design, code and documentation) from the repository should require the least time and effort.

Chapters 5, 6, 7 and 8 discuss the structural, functional, behavioral and multi-view similarity assessment techniques used during retrieval. In each of these chapters, the matching techniques and similarity measures are described. Matching refers to mapping an entity in one model to another entity of the same type in the other model to be compared. Once a pair of entities has been mapped, a similarity scoring algorithm can be used to compute their degree of similarity. An entity could be a class or an entire diagram. Matching makes use of one or more of the heuristic search techniques described in Section 4.5.

4.4 Concept of Operation

This section gives an overall picture of our reuse system by describing the sequence of steps needed to reuse software contained in a repository. The prerequisite for using the reuse system is that the requirement specification for the query should contain at least one of the following UML diagrams: class diagram, sequence diagram(s) or state machine diagram(s). In addition, the requirement specification for each project in the repository should have at least one type of UML diagram in common with the query. It is noteworthy that the requirement specifications for repository projects act as indexes to the projects. Figure 4.1 illustrates the steps involved in the reuse process. These steps are described below:

1. The user presents a query requirement specification which contains at least one of the following diagrams: class diagram, sequence diagram(s) or state machine diagram(s).
2. The metadata for the query are computed.
3. Pre-computed metadata for each project in the repository are retrieved. By comparing the metadata of query and repository projects, a similarity score is computed between the query and repository projects.
4. Based on the similarity scores, a list of repository projects that are potentially similar to the query is created.
5. The requirement specifications (comprising class diagram, sequence diagrams and state machine diagrams) for each shortlisted repository project are retrieved from the repository.

6. The degree of similarity between the query and shortlisted repository projects are computed by comparing their requirement specifications.
7. The most similar existing project to the query is determined
8. A copy of all the artifacts for the most similar repository project is returned to the user. These artifacts include design, source code, documentation and test data.
9. The user modifies the artifacts to suit the needs of the new software system being developed. S/he stores the modified artifacts in the repository, so that the new project can be reused in the future.

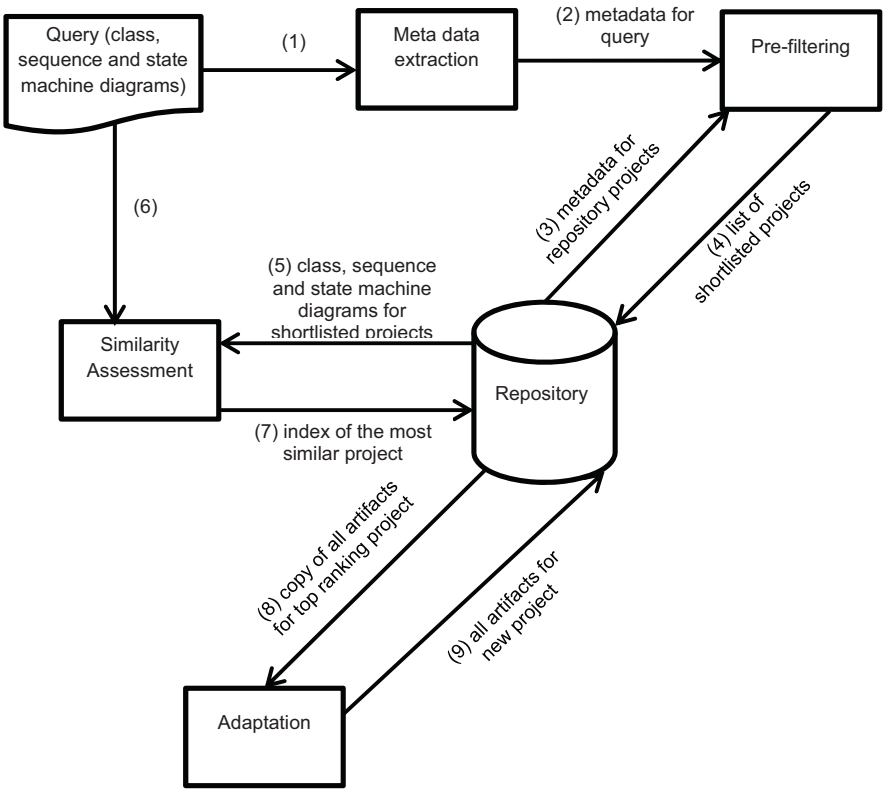


Figure 4.1: Concept of operation of reuse system

4.5 Heuristic Search Techniques

Taking into consideration the issues raised in Section 2.2, the problem of matching during multi-view similarity assessment can be viewed as a constrained mapping of model entities (e.g., classes and sequence diagrams) of a query requirement specification to particular counterparts in the repository such that the instantiations (i.e., the mappings from query specifications to repository specifications) have the least conflicts. In essence, this problem is a constraint satisfaction combinatorial optimization problem. Accordingly, finding a mapping that produces optimal similarity of UML artifacts represents a NP-hard problem.

Nature inspired heuristic search algorithms have been used to solve wide range of combinatorial optimization problems, including NP-hard problems such as the travelling salesman problem [49]. In order to use any of the heuristic search algorithms, three important issues need to be considered [50]: encoding a solution, defining transformations and determining how good a solution is. Several heuristic search algorithms are described next:

4.5.1 Hill Climbing (HC)

Hill climbing is an iterative search algorithm that begins with a random solution and then searches through its neighbors for a better solution. Hill climbing does not look ahead beyond immediate neighbors of the current state thus it is also referred to as greedy local search. The algorithm may get stuck due to a local maxima, plateau or ridge [50, 51].

4.5.2 Tabu Search (TS)

In Tabu search, the space of all possible solutions is searched in a sequence of moves from one possible solution to the best available alternative. To avoid being stuck at a suboptimal solution and drifting from the global optimum, some moves are classified as forbidden or *tabu* (taboo). The list of *tabu* moves is formed using short-term and long-term memory of previous unpromising moves [52].

4.5.3 Simulated Annealing (SA)

Simulated annealing imitates the annealing process in metallurgy used when cooling of metals needs to be stopped at given points and then warmed a bit before the cooling process resumes. Simulated annealing begins with a point x in the search space which is chosen heuristically or randomly. The cost value c, given by cost function E, of point x is then calculated. Next a neighboring value x_1 is searched and its cost c_1 is computed. If $c_1 < c$, the search moves onto x_1. However, even though $c_1 > c$, there is still a chance, given by probability p, that search is allowed to continue to a solution with a bigger cost. The value of p depends on the current temperature and the change in cost function [50].

4.5.4 Genetic Algorithm (GA)

Genetic algorithms were invented in the 1960s by John Holland. Genetic algorithm begins with a set of randomly generated candidate solutions (chromosomes) forming the population. Each chromosome is made up of a set of genes representing some property of the solution. A fitness function is used to measure the goodness of a solution. During crossover (i.e., reproduction), genes are exchanged between pairs of parent chromosomes to form new chromosomes. With a small probability, an offspring is subjected to

mutation, where bits of information in the chromosome are changed to prevent stagnation of the search [50, 51].

4.5.5 Particle Swarm Optimization (PSO)

Particle Swarm Optimization mimics the group behavior of animals such as bird flocking and fish schooling. PSO optimizes a problem having a population (swarm) of candidate solutions (particles). The movements of the particles are guided by their current velocity, their best known position in the search-space and the entire swarm's best known position. When improved positions are discovered, they guide the movements of the swarm. The process is repeated and by doing so it is hoped, but not guaranteed, that a satisfactory solution will eventually be discovered. PSO is similar to GA in some aspects: (i) they are population-based algorithms; (ii) they start with a randomly generated population, and (iii) they update the population and search for the optimum with random techniques [53].

4.5.6 Cuckoo Search Algorithm (CSA)

Cuckoo search is a relatively new heuristic search algorithm that mimics the aggressive reproduction strategy of cuckoos. Some cuckoo species engage in brood parasitism by laying their eggs in other birds' nests. When host birds discover eggs that are not theirs, they may throw away the eggs or abandon their nests and build new nests elsewhere. Each egg in a nest represents a solution, while a cuckoo egg represents a new solution. The aim of the search is to replace not-so-good solutions in the nests with new and potentially better solutions. CSA works based on three rules. (i) Each cuckoo lays an egg at a time and drops it into a randomly chosen nest. (ii) The best nests (i.e. those with high quality solutions) move over to the next generation. (iii) The number of nests is fixed,

and each host discovers a foreign egg with a certain probability. Upon discovery, it may throw away the egg or abandon the nest and build new solutions [49].

Chapter 5

Structural Similarity Assessment

Class diagrams are arguably the most popular of all the 14 UML diagrams. They depict the structure of a system by showing the system's classes and the relationships between them. This chapter describes how the degree of similarity between software systems can be determined by comparing their class diagrams. Two forms of structural similarity assessment are presented: shallow structural similarity assessment, which compares class diagrams by considering only the relationships between classifiers (i.e., classes and interfaces); and deep structural similarity assessment, which mainly relies on information contained within classifiers to determine class diagram similarity.

In order to assess the shallow similarity between class diagrams, they are converted to directed graphs which are then compared. In essence, class diagrams' similarity is formulated as a graph matching/similarity problem. Our similarity assessment technique considers the relationships between classifiers without looking at the internal details of each classifier such as attributes and methods. The primary motivation for doing this is to allow reuse to take place across domains. This idea is also applied in analogy; the structure mapping theory is an analogical mapping technique which allows knowledge to be mapped in two domains by considering relational commonalities of objects in the domains regardless of the objects involved in the relationships.

It is worth noting that Rufai [5] has also used the terms *shallow similarity* and *deep similarity* while comparing class diagrams. However, the work presented in this chapter differs from Rufai's work as follows: Firstly, shallow similarity assessment is carried out in this chapter by comparing the relationships between classifiers in two class diagrams, whereas shallow similarity scores are computed in [5] by comparing the names of classifiers in two class diagrams. Secondly, deep similarity value is computed in this chapter by comparing the constituents of classifiers such as their method lists, attribute lists, the names and data types of attributes, and the names and signatures of methods. On the other hand, Rufai [5] uses only method names and attribute names to compute deep similarity scores.

Section 5.1 discusses shallow similarity assessment, and covers the graphical representation of class diagrams, the structural similarity measure, and the use of heuristic search techniques for matching classifiers. In Section 5.2, our method of deep similarity assessment is presented. Section 5.3 summarizes the entire chapter.

5.1 Shallow Similarity Assessment of Class Diagrams

This section describes how the structural similarity between class diagrams can be computed based on the relationships between classifiers in the class diagram.

5.1.1 Graph Representation of Class Diagrams

UML class diagrams can be converted to labeled directed graphs in which classes are represented by nodes, and the relationships between them are represented as edges of the graph. Moreover, edges are labelled to specify which UML relationship they represent. With this representation in mind, the problem of matching a query class diagram to

another class diagram in the repository becomes that of graph matching. Figure 5.1 shows two sample class diagrams *A* and *B*. A graph representation of *A* can be seen in Figure 5.2. An adjacency matrix representation of the graph is also shown in Table 5.1. Rather than containing zeroes and ones, the entries of the matrix show the types of relationships represented by edges of the graph.

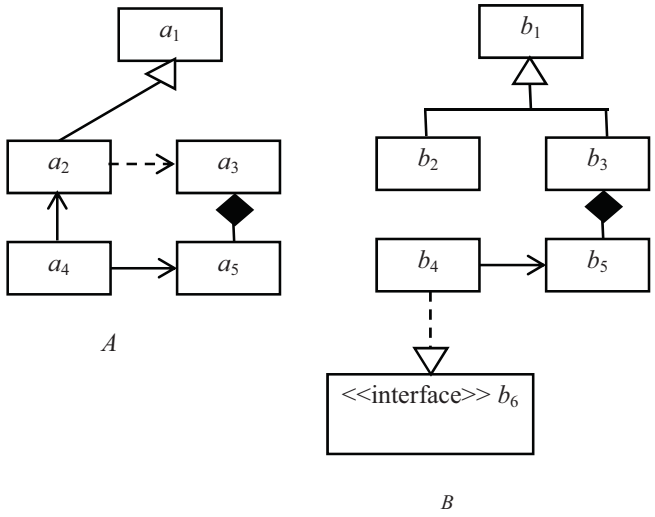


Figure 5.1: Two sample class diagrams *A* and *B*

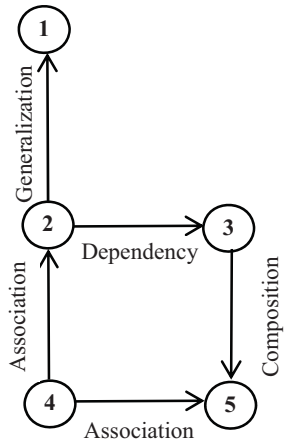


Figure 5.2: Graph representation of diagram *A* of Figure 5.1

Table 5.1: Adjacency matrix of diagram *A* of Figure 5.1

	a_1	a_2	a_3	a_4	a_5
a_1	None	None	None	None	None
a_2	Generalization	None	Dependency	None	None
a_3	None	None	None	None	Composition
a_4	None	Association	None	None	Association
a_5	None	None	None	None	None

5.1.2 Similarity Measure

In order to determine the shallow structural similarity score from two adjacency matrices, we define a Difference Matrix (*Diff*), whose entries represent the level of dissimilarity between the various types of class diagram relationships. The $(i, j)^{th}$ entry of the matrix is a measure of the dissimilarity between the i^{th} type of relationship and the j^{th} type of relationship. A value of 1 indicates that the two relationships are not similar at all, whereas 0 indicates that the relationships are identical (hence the diagonal entries of the matrix are all zeroes). The entries of this matrix can be filled by gathering information from software professionals [39]. Table 5.2 (adapted from [39]) shows *Diff*. The last row labeled 'NO' shows the level of dissimilarity between having no relationship between two classes (that is no edge connecting the vertices) and having a relationship between the two classes. Since the main objective of retrieving class diagrams is to reuse them, the entries in *Diff* should be proportional to the amount of effort required to convert one type of relationship to another after retrieving a class diagram from the repository.

Let *AdjA* and *AdjB* be adjacency matrices of *A* and *B*, whose degree of similarity is to be determined. Assume *A* has *na* classifiers, while *B* has *nb* classifiers ($na \leq nb$). Let *P* be a permutation vector that maps all *na* classes of *A* to *na* classes of *B*. In addition, let $AdjB_P$ be a *na* X *na* adjacency matrix that contains only the relationships between classifiers of

B listed in *P*. For example, *P* = (2, 3, 5) implies that the 1st, 2nd and 3rd classes of *A* are mapped to the 2nd, 3rd and 5th classes of *B*. Furthermore, $AdjB_P$ is a 3 X 3 matrix showing the relationships between the 2nd, 3rd and 5th classes of *B*. The degree of similarity between *A* and *B* is given in Eq. (5.1).

$$sim(A,B) = \frac{\sum_{i=1}^{na}\sum_{j=1}^{na} Diff(AdjA(i,j), AdjB_P(i,j))}{nr} + \frac{\alpha(nb-na)}{na} \tag{5.1}$$

where *nr* is the number of times there is at least one relationship at corresponding entry positions in *AdjA* or $AdjB_P$. $\alpha \in [0, 1]$ is a weight that determines how the unmapped classifiers in *B* affect the degree of similarity. When α is zero, the degree of similarity between *A* and *B* is zero (indicating maximum similarity) whenever *B* subsumes *A*. However, a large value of α causes the value of *sim*(*A, B*) to increase when *nb* > *na*.

Table 5.2: *Diff* matrix (adapted from [39])

	AS	**AG**	**CO**	**DE**	**GE**	**RE**	**IR**	**NO**
AS	0.00	0.11	0.11	0.45	0.45	0.66	0.77	1.00
AG	0.11	0.00	0.11	0.45	0.45	0.66	0.77	1.00
CO	0.11	0.11	0.00	0.45	0.45	0.66	0.77	1.00
DE	0.49	0.49	0.49	0.00	0.28	0.21	0.32	1.00
GE	0.49	0.49	0.49	0.28	0.00	0.49	0.60	1.00
RE	0.83	0.83	0.83	0.34	0.62	0.00	0.11	1.00
IR	1.00	1.00	1.00	0.51	0.79	0.17	0.00	1.00
NO	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00

AS = ASSOCIATION, AG = AGGREGATION, CO = COMPOSITION, DE = DEPENDENCY, GE = GENERALIZATION, RE = REALIZATION, IR = INTERFACE REALIZATION, NO = NO RELATIONSHIP

We now turn our attention to determining analytically if the similarity measure of Eq. (5.1) is a similarity metric. Similarity (or dissimilarity) measures can be thought of as explaining distances in metric space. Similarity measures which satisfy certain metric

axioms are referred to as similarity metrics [54]. The metric axioms can be stated as follows:

(i)	$sim(a, a) = sim(b, b)$	(self-similarity)
(ii)	$sim(a, b) \geq sim(a, a)$	(minimality)
(iii)	$sim(a, b) = \text{sim}(b, a)$	(symmetry)
(iv)	$sim(a, b) + sim(b, c) \geq sim(a, c)$	(triangle inequality)

In the following paragraphs, we prove that the similarity measure of Eq. (5.1) satisfies the first two metric axioms.

Self-similarity: Since the diagonal entries of *Diff* are zero and corresponding entries of the adjacency matrices of identical class diagrams are the same, the numerator of the first fraction in Eq. (5.1) is zero. Furthermore, identical class diagrams have the same number of nodes so the numerator of the second fraction in Eq. (5.1) is zero. It therefore follows that $sim(A, A) = sim(B, B) = 0$.

Minimality: There are two cases to consider:

Case 1: if $A = B$, it follows that $sim(A, B) = sim(A, A) = 0$ from the first axiom.

Case 2: if $A \neq B$, either or both of the following conditions is true: (i) there is at least one pair of classifiers whose corresponding edges differ in *AdjA* and *AdjB*. Since the non-diagonal entries of *Diff* are nonzero, the numerator of the first fraction in Eq. (5.1) is greater than zero (ii) *A* and *B* have different number of nodes, thus the numerator of the second fraction in Eq. (5.1) is greater than zero. From (i) and/or (ii), $sim(A, B) \in (0, 1]$.

Thus, $sim(A, B) \geq sim(A, A)$

Symmetry: Because *Diff* is asymmetric, the similarity measure of Eq. (5.1) does not satisfy symmetry. It is understandable that *Diff* is asymmetric, since it reflects the amount of effort required to convert one type of relationship to another.

Triangular inequality: We have not been able to prove that Eq. (5.1) satisfies triangular inequality, which is the most difficult of the axioms to prove [54]. Accordingly, we shall refer to the formula in Eq. (5.1) as a similarity measure rather than a metric.

5.1.3 Computation of Classifiers' Similarity Matrix

In this section, we describe a method of computing pairwise similarity between classifiers (i.e., classes and interfaces) in two class diagrams. The similarity values are contained in a classifiers' similarity matrix *M*, which will be utilized during matching. (see Section 5.1.4). Each classifier is represented by a set of features in a 14 dimensional vector space. Each dimension of the vector indicates how many of the different UML class diagram relationships begin with or end at a class. The different features of a classifier *c* are stored in a feature vector $fc = <fc_1, fc_2, fc_3 \ldots fc_{14}>$.

Each feature is described in Table 5.3. Table 5.4 shows the features of classifiers belonging to diagrams *A* and *B* of Figure 5.1. From Table 5.4, it can be seen that b_3 is a client in one composition relationship; it is also a child in one generalization relationship. The similarity between two classifiers is the Euclidean distance between the classifiers' feature vectors. Table 5.5 shows matrix *M* containing the pairwise similarity between classifiers in *A* and *B*. It can be inferred from Table 5.5 that a_5 and b_5 are involved in exactly the same types of relationships because their similarity value is zero.

Table 5.3: Description of features of each classifier

Dimension	fc_1	fc_2	fc_3	fc_4	fc_5	fc_6	fc_7	fc_8	fc_9	fc_{10}	fc_{11}	fc_{12}	fc_{13}	fc_{14}
Number of times the classifier appears as:	Association Client	Association Supplier	Aggregation Client	Aggregation Supplier	Composition Client	Composition Supplier	Dependency Client	Dependency Supplier	Generalization Child	Generalization Parent	Realization Client	Realization Supplier	Interface Realization Client	Interface Realization Supplier

Table 5.4: Feature vectors of classifiers in *A* and *B*

Dimension	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a_1	0	0	0	0	0	0	0	0	0	1	0	0	0	0
a_2	0	1	0	0	0	0	1	0	1	0	0	0	0	0
a_3	0	0	0	0	1	0	0	1	0	0	0	0	0	0
a_4	2	0	0	0	0	0	0	0	0	0	0	0	0	0
a_5	0	1	0	0	0	1	0	0	0	0	0	0	0	0
b_1	0	0	0	0	0	0	0	0	0	2	0	0	0	0
b_2	0	0	0	0	0	0	0	0	1	0	0	0	0	0
b_3	0	0	0	0	1	0	0	0	1	0	0	0	0	0
b_4	1	0	0	0	0	0	0	0	0	0	0	0	1	0
b_5	0	1	0	0	0	1	0	0	0	0	0	0	0	0
b_6	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Table 5.5: Classifiers' similarity matrix *M*

	b_1	b_2	b_3	b_4	b_5	b_6
a_1	1.00	1.41	1.73	1.73	1.73	1.41
a_2	2.65	1.41	1.73	2.24	1.73	2.00
a_3	2.45	1.73	1.41	2.00	2.00	1.73
a_4	2.83	2.24	2.45	1.41	2.45	2.24
a_5	2.45	1.73	2.00	2.00	0.00	1.73

5.1.4 Matching of Classifiers using heuristic search algorithms

Exhaustively searching for an optimal value of permutation vector *P* results in examining a huge search space. There are $_{nb}P_{na} = \frac{nb!}{(nb-na)!}$ possible values of *P*. For example, if *na* =

50 and nb = 60, an exhaustive search for the best value of P will involve 2.29 x 10^{75} comparisons.

The problem of finding a suitable value of P that results in an optimal (i.e., smallest) similarity value between A and B is a combinatorial optimization problem. This section describes the use of two heuristic search algorithms namely; GA and CSA to obtain a suitable value of P in order to compute the similarity between A and B. The inputs to the heuristic search algorithms are $AdjA$, $AdjB$ and M. The outputs of the algorithm are the values of $sim(A, B)$ and P.

Matching using GA

This section describes the GA for obtaining a suitable value of P. The GA is similar to that used for graph matching by Wang and Ishii [55]. First, an initial population of chromosomes is constructed. During each iteration, pairs of selected individuals are combined to form new offspring by applying a crossover operator. After crossover, a mutation operator is applied to randomly chosen chromosomes to prevent the search from being stuck in a local optimum. Next, duplicated individuals are mutated until each individual in the population is unique. The algorithm iterates until one of the termination conditions is met. Detailed descriptions of the GA are provided in what follows.

Chromosome Encoding

Each chromosome is a row vector of the same form as P. In other words, the number in the i^{th} gene indicates which classifier of B is mapped to the i^{th} classifier of A. Figure 5.3 shows a possible chromosome encoding for computing the similarity score of the two class diagrams in Figure 5.1.

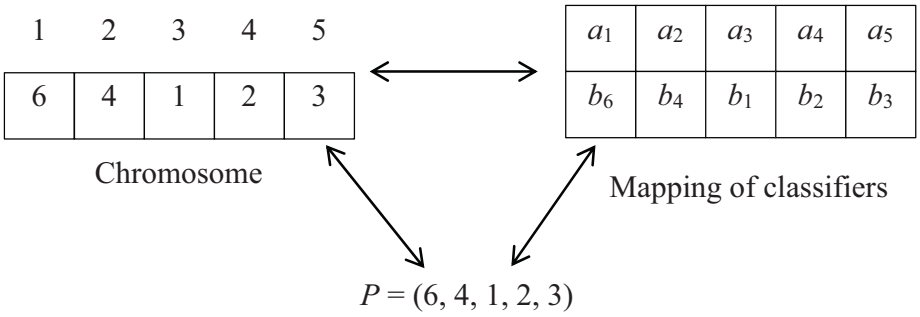


Figure 5.3: Chromosome encoding for comparing two class diagrams

Population Initialization.

There are a fixed number (n) of individuals in each generation. At the beginning of GA, m individuals are formed in two steps: (i) Munkres' allocation algorithm [56] is applied to similarity matrix M to obtain a mapping of nodes for the first individual; (ii) an additional m - 1 individuals are constructed by swapping any two randomly selected genes in the first individual. The remaining $n - m$ individuals in the population are formed by randomly choosing values for their genes.

Figure 5.4 illustrates population initialization for $m = 3$. The first individual is formed by applying Munkres' algorithm on the similarity matrix of Table 5.5. The second and third individuals are formed by swapping two randomly selected genes in the first individual. The swapped genes are shown in grey in Figure 5.4. Finally, the genes for the remaining individuals in the population are randomly filled.

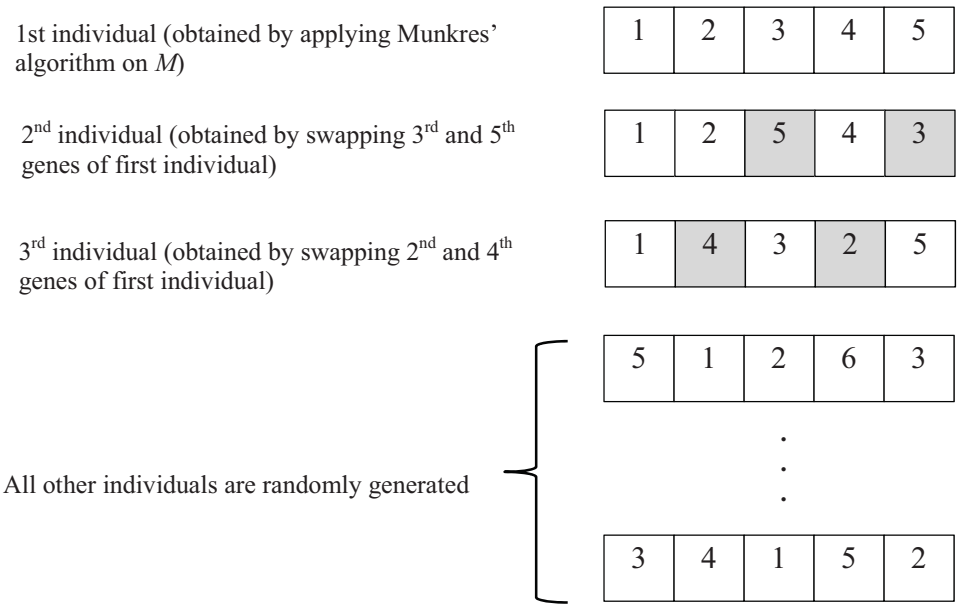


Figure 5.4: Population initialization

Fitness Values

The fitness value of a gene is obtained from *M*. For example, the fitness value of the i^{th} gene of a chromosome is $M(i, j)$, where j is the value contained in the i^{th} gene. The fitness of an individual is computed using Eq. (5.1), while the fitness of a population is the minimum of the fitness values of individuals in the population.

Termination Conditions

The GA terminates when any of the following conditions is satisfied: the population fitness reaches 0 (i.e., maximum degree of similarity between *A* and *B* is obtained); a pre-set maximum number of iterations is reached; or the population fitness does not improve within a given number of iterations.

Selection

Each of the n individuals is selected for crossover. The individuals are sorted in increasing order of fitness values. Since a crossover of two parents results in one offspring, n pairs of individuals are selected for crossover in the following manner: the crossover operator is applied to the i^{th} and $(i+1)^{th}$ individuals ($1 \leq i \leq \lceil n/2 \rceil$) resulting in $\lceil n/2 \rceil$ individuals of the next population. Furthermore, the crossover operator is applied to the j^{th} and $(n + 1 - j)^{th}$ individuals ($1 \leq j \leq \lfloor n/2 \rfloor$) to generate the remaining $\lfloor n/2 \rfloor$ individuals of the next population. This method of selecting individuals for crossover leads to a higher average fitness value for the next generation and/or the creation of individuals with higher fitness values [55]. Figure 5.5 illustrates how individuals are selected.

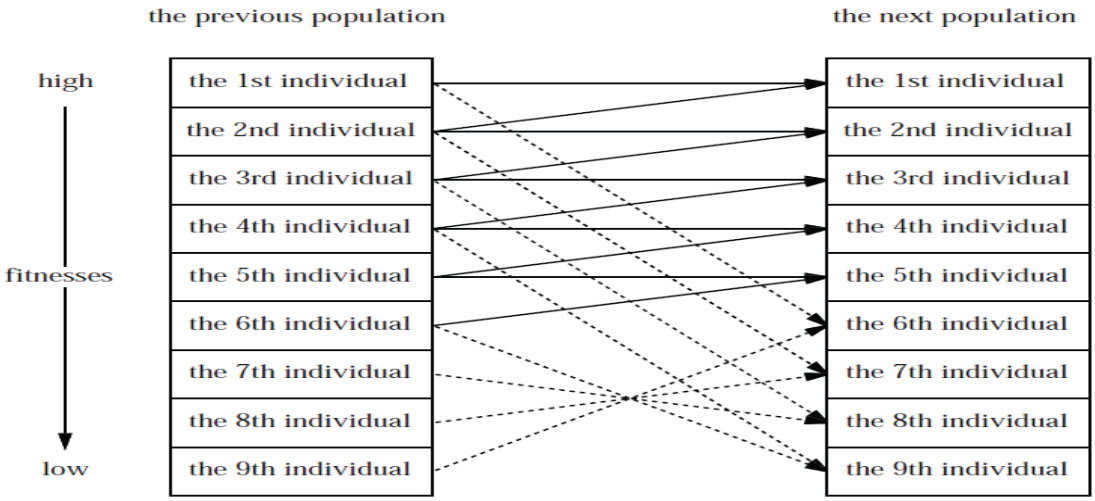


Figure 5.5: Selection operation [57]

Crossover

Our crossover operator is similar to that used in [55]. Let the two parents selected for crossover be P_1 and P_2, where the fitness of P_1 is better (i.e., has lower value) or equal to

that of P_2. The crossover operation entails three steps. Firstly, the genes of P_1 which are fitter than corresponding genes of P_2 are copied to the offspring. Secondly, the genes of P_2 which are fitter than corresponding genes of P_1 are copied to the offspring, provided they are not already present in the offspring. The latter condition in the second step ensures that invalid chromosomes are not formed. Finally, the remaining genes of the offspring are randomly filled with values that are not already present in the offspring. In this way, it is expected that fitter offspring will be produced because they combine the good parts (genes) of both their parents. However, if the fitness of the offspring is worse than that of P_1, the offspring is discarded and replaced by P_1. Steps of the crossover operation are shown in Figure 5.6. The grey-colored cells in the figure indicate the genes that are filled in each step.

Before crossover	**Fitness**	0.1	0.4	0.5	0.2	0.3	0.6	0.2
	P_1	1	2	3	4	5	6	7
	P_2	6	3	5	2	4	7	1
	Fitness	0.6	0.5	0.3	0.4	0.2	0.2	0.1
1. Copy fitter genes from P_1	**Offspring**	1	2		4			
2. Copy fitter genes from P_2 that are unused	**Offspring**	1	2	5	4		7	
3. Randomly fill remaining positions with unused genes	**Offspring**	1	2	5	4	3	7	6

Figure 5.6: Crossover operation

Mutation

In order to ensure diversity of the population and prevent GA from being trapped in a local optima, there is a small probability that each gene in the population is mutated. Mutation results in the swapping of two genes in a chromosome, or the replacement of one gene with another gene that is absent from the chromosome. Figure 5.7 illustrates the

mutation operation. In Figure 5.7 (a), the second and fourth genes are swapped during mutation. Figure 5.7 (b) shows that the third gene is replaced by a value that is not already in the chromosome. Note that the later type of mutation is possible only when the two class diagrams being compared have different number of classifiers.

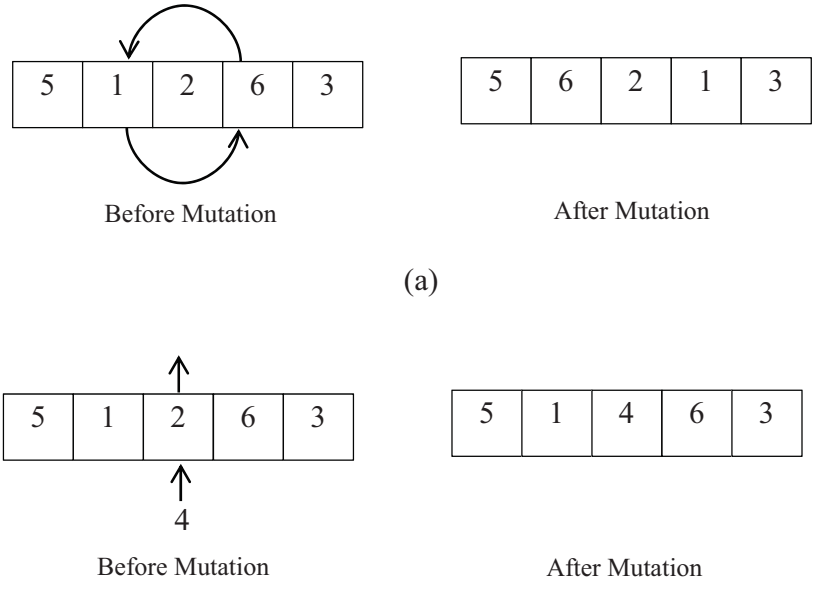


Figure 5.7: Mutation by (a) swapping two genes (b) replacing a gene with a value not found in the chromosome

Uniqueness of Individuals in a Population

At the end of each generation, duplicate individuals in the population are eliminated by repeatedly mutating one of the replicas until it becomes distinct from all other individuals. One reason for having identical individuals in the population is that at the end of crossover, the fitter parent (P_1) replaces the offspring if the latter is not as fit as the former. Because the fittest parents are selected for crossover twice (see Figure 5.5) it is possible that a very fit parent is returned as the offspring after both crossover operations.

Matching using CSA

The algorithm for cuckoo search is shown in Figure 5.8. Each nest is of the same form as *P* since a nest is analogous to a chromosome in GA. CSA begins by creating an initial population of nests in the same way as GA. Next, nests are partitioned into top and bottom nests based on their fitness values.

Each nest in the bottom partition is abandoned and replaced by performing a Lévy flight from that nest. Lévy flight is a random walk in the search space using a random step length obtained from a Lévy distribution, and it is known to improve the performance of CSA [58]. In this work, we carry out Lévy flight by randomly swapping two values in a nest (i.e., similar to mutation in GA) several times according to the random step length.

Next, fitness values of all nests are recomputed. Again, nests are partitioned into two sets based on their fitness values: top nests and bottom nests.

For each top nest *i,* another top nest *j* is randomly selected. If *i* and *j* are the same, a new nest *k* is formed by performing Lévy flight from nest *i*. However, if nests *i* and *j* differ, the distance *dist* between them is obtained by dividing their hamming distance (i.e., the sum of the absolute values of their differences) by the golden ratio which is approximately 1.62 [58]. A new nest *k* is formed by randomly moving *dist* steps from the current nest. Next, a nest *l* is randomly selected from the entire population. If nest *k* is fitter than nest *l*, it replaces it.

The process is repeated until one of the terminating conditions is met. These conditions are exactly the same ones used during GA. It is important to note that during all flights (steps 5, 10 and 14), the new nest is taken as the fittest nest found along the way, rather

than the last nest on the path. In other words, a local search is being performed along the way.

1. Generate initial population
2. While termination conditions have not been met, repeat
    3. Sort nests according to their fitness values using Eq. (5.1)
    4. Partition nests into two sets: top and bottom nests
    5. For each bottom nest, perform Lévy flight and replace with new value
    6. Sort and partition nests again
    7. For each top nest *i*, do
        8. Randomly select another top nest say, *j*
        9. if *i* = *j*
            10. Form a new nest *k* by performing Lévy flight from nest *i*
        11. else
            12. Find *hamming_distance* between nest *i* and nest *j*
            13. *dist* = *hamming_distance*/*golden_ratio*
            14. Form a new nest *k* by moving *dist* steps from nest *i*
        15. end if
        16. Randomly choose a nest *l* from the entire population
        17. If fitness of nest *l* is worse than that of nest *k*, replace it with nest *k*
    18. End for

Figure 5.8: Algorithm for cuckoo search

5.2 Deep Similarity Assessment

Deep similarity assessment is a very detailed comparison of the constituent parts of a class diagram. This section describes how the deep similarity score is computed. In the reminder of this section, similarity measures are presented in a bottom-up manner,

starting from the similarity measure for comparing two entities that each have a name and data type (referred to as a name-type pair), and culminating in the similarity measure for two class diagrams. Figure 5.9 shows the hierarchy of computations required to produce a deep similarity score. It is worth mentioning that in this section of the dissertation, higher similarity values show a high degree of similarity. In every other part of the dissertation, we represent high degrees of similarity by small similarity scores.

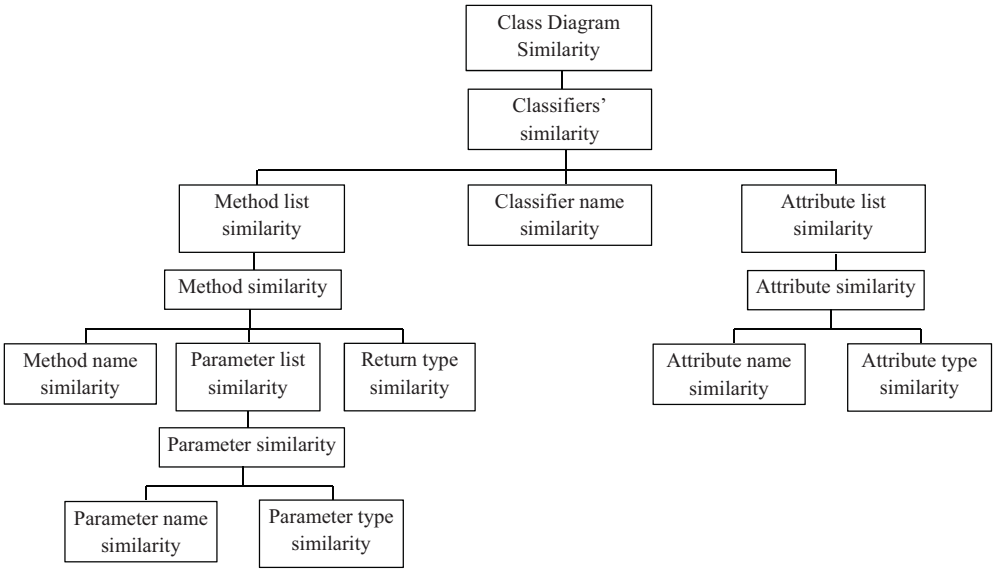


Figure 5.9: Hierarchy of computations for deep similarity assessment

5.2.1 Similarity between two strings

The edit distance between two strings is the minimum number of single character edits (insertions, substitutions or deletions) required to transform one string to another. For example, the edit distance between 'alters' and 'water' is three because the former string can be transformed to the latter in (at least) three steps: insert 'w' at the beginning; delete 'l'; and delete 's'. The degree of similarity between two strings S_1 and S_2 can be computed from their edit distance as follows:

$$ED(S_1, S_2) = 1 - \frac{\text{edit distance}(S_1, S_2)}{\max(|S_1|, |S_2|)} \tag{5.2}$$

where *edit distance* is a function that returns the edit distance between two strings. $|S_i|$ returns the number of characters in string S_i. *max* is a function that returns the larger of two numbers. The similarity value returned by *ED* ranges from zero (for highly dissimilar strings) to one (for identical strings).

5.2.2 Similarity between name-type pairs

A name-type pair may refer to an attribute name and its data type, or a parameter name and its data type. The similarity between two name-type pairs NT_1 and NT_2 can be found using Eq. (5.3).

$$SNT(NT_1, NT_2) = w_1 \cdot ED(NT_{1\_name}, NT_{2\_name}) + (1 - w_1) \cdot ED(NT_{1\_datatype}, NT_{2\_datatype}) \tag{5.3}$$

w_1 is a weight that determines the relative importance of name and data type similarities. NT_{i_name} and NT_{i_datatype} denote the name and data type for the name-type NT_i (i = 1, 2).

5.2.3 Similarity between attribute lists

An attribute list is the set of attributes belonging to a class. As shown in Eq. (5.4), the similarity between the attribute lists of two classes can be computed from the pairwise similarity of their attributes. The similarity between attributes is computed from their names and data types using Eq. (5.3).

$$SAL(AL_1, AL_2) = \frac{\max\_total\_sim\left(\begin{bmatrix} SNT(AL_{11}, AL_{21}) & \ldots & SNT(AL_{11}, AL_{2|AL_2|}) \\ \ldots & \ldots & \ldots \\ SNT(AL_{1|AL_1|}, AL_{21}) & \ldots & SNT(AL_{1|AL_1|}, AL_{2|AL_2|}) \end{bmatrix}\right)}{\max(|AL_1|, |AL_2|)} \tag{5.4}$$

Where AL_1 and AL_2 are two attribute lists having $|AL_1|$ and $|AL_2|$ attributes, respectively. AL_{ij} denotes the j^{th} attribute of AL_i (i = 1, 2). *max_total_sim* is the maximum total similarity obtained by applying Munkres' allocation algorithm on the pairwise similarities of attributes. The denominator of the fraction is the maximum of the number of attributes in the two attribute lists. When the attribute lists contain different number of attributes, this has the effect of reducing the similarity value (and thus, the degree of similarity).

5.2.4 Similarity between methods

The similarity between methods is made up of three parts: similarity between their names; return types; and parameter lists. Similarity between parameter lists is computed in the same way as that between attribute lists, except that attributes in Eq. (5.4) are replaced by parameters. The degree of similarity between two methods M_1 and M_2 can be computed using Eq. (5.5);

$$SM(M_1, M_2) = w_2 \cdot ED(M_{1\_name}, M_{2\_name}) + w_3 \cdot ED(M_{1\_returntype}, M_{2\_returntype}) + (1 - w_2 - w_3) \cdot \frac{\max\_total\_sim\left(\begin{bmatrix} SNT(P_{11}, P_{21}) & \dots & SNT(P_{11}, P_{2|M_2|}) \\ \dots & \dots & \dots \\ SNT(P_{1|M_1|}, P_{21}) & \dots & SNT(P_{1|M_1|}, P_{2|M_2|}) \end{bmatrix}\right)}{\max(|M_1|, |M_2|)} \quad (5.5)$$

Where $|M_1|$ and $|M_2|$ are the number of input parameters for M_1 and M_2, respectively. P_{ij} is the j^{th} input parameter of M_i (i = 1, 2). M_{i_name} and $M_{i\_returntype}$ refer to the name and return type of M_i, respectively (i = 1, 2). w_2 and w_3 are constants that determine the relative weights of the three components that make up the similarity of methods.

5.2.5 Similarity between method lists

The method list for a class is the set of methods for that class. As shown in Eq. (5.6), the similarity between two method lists is computed by applying Munkres' algorithm on the pairwise similarities of the methods contained in both lists.

$$SML(ML_1, ML_2) = \frac{\max\_total\_sim\left(\begin{bmatrix} SM(ML_{11}, ML_{21}) & \dots & SM(ML_{11}, ML_{2|ML_2|}) \\ \dots & \dots & \dots \\ SM(ML_{1|ML_1|}, ML_{21}) & \dots & SM(ML_{1|ML_1|}, ML_{2|ML_2|}) \end{bmatrix}\right)}{\max(|ML_1|, |ML_2|)} \quad (5.6)$$

Where ML_1 and ML_2 are two method lists having $|ML_1|$ and $|ML_2|$ methods, respectively. ML_{ij} denotes the j^{th} method of ML_i (i = 1, 2).

5.2.6 Similarity between classifiers

The similarity between two classifiers is computed as a weighted sum of the degree of similarity of the classifiers' names, their attributes lists and their method lists using Eq. (5.7):

$$SC(C_1, C_2) = w_4 \cdot SML(C_{1\_methodlist}, C_{2\_methodlist}) + w_5 \cdot SAL(C_{1\_attributelist}, C_{2\_attributelist}) + (1 - w_4 - w_5) \cdot ED(C_{1\_name}, C_{2\_name}) \quad (5.7)$$

Where C_{i_name}, $C_{i\_methodlist}$ and $C_{i\_attributelist}$ are the name, list of methods and list of attributes, respectively for classifier C_i (i = 1, 2). w_4 and w_5 determine the relative significance of the three constituent parts of the similarity value.

5.2.7 Similarity between class diagrams

The degree of similarity between two class diagrams CD_1 and CD_2 is determined by using Munkres' algorithm to determine the best overall similarity from the pairwise

similarities of classifiers in both class diagrams. Eq. (5.8) can be used to compute the similarity between class diagrams:

$$SCD(CD_1, CD_2) = \frac{\max\_total\_sim\left(\begin{bmatrix} SC(CD_{11}, CD_{21}) & \ldots & SC(CD_{11}, CD_{2|CD_2|}) \\ \ldots & \ldots & \ldots \\ SC(CD_{1|CD_1|}, CD_{21}) & \ldots & SC(CD_{1|CD_1|}, CD_{2|CD_2|}) \end{bmatrix}\right)}{\max(|CD_1|, |CD_2|)} \quad (5.8)$$

Where CD_{ij} is the j^{th} classifier of CD_i and $|CD_i|$ is the number of classifiers contained in CD_i ($i = 1, 2$).

In the remainder of this section, we prove that the deep similarity measure of Eq. (5.8) satisfies three of the four metric axioms. Note that Eq. (5.8) is computed from a linear combination of three building blocks: the edit distance similarity function *ED*; *max_total_sim* which, given a matrix, returns an optimal similarity by applying Munkres algorithm; and *max*, which simply returns the larger of two numbers.

Symmetry: $SCD(a, b) = SCD(b, a)$ since the three basic functions used to compute Eq. (5.8) are all symmetric. Firstly, applying Munkres algorithm on a matrix *M* gives the same optimal similarity as applying the algorithm on the transpose of *M*. Secondly, *ED*(…) is computed from the edit distance of two strings. The edit distance is the minimum number of operations needed to convert one string to another, so it is not affected by the order of the two strings. Lastly, it is well known that finding the maximum of two numbers is a symmetric operation.

Self-similarity: If two class diagrams are identical, the numerator of Eq. (5.8) is computed by applying Munkres algorithm on a square matrix, say *M*. There is at least one entry in each row and each column of *M* that contains 1, indicating maximum similarity

between classifiers. Therefore, the value returned by Munkres algorithm is equal to the number of rows of *M*. Moreover, since identical class diagrams have equal number of classifiers, the denominator of Eq. (5.8) is also the same as the number of rows of *M*. Thus, Eq. (5.8) returns an optimal similarity value of 1 for any two identical class diagrams.

Minimality: Since higher similarity values indicate better degree of similarity, we prove *maximality* instead. There are two cases to consider:

Case 1: if $a = b$, it follows that $SCD(a, b) = SCD(a, a) = 1$ from the axiom of self-similarity.

Case 2: if $a \neq b$, assume the numerator of Eq. (5.8) is $max\_total\_sim(M) / max(m_1, m_2)$ where M is a m_1 x m_2 matrix of pairwise similarity values. Either or both of the following conditions is true: (i) there is at least one classifier which cannot be mapped to an identical classifier in the other class diagram (i.e., at least one row or one column does not contain a value of one), thus $max\_total\_sim(M) < max(m_1, m_2)$, resulting in a similarity value that is less than one. (ii) $m_1 \neq m_2$, thus $max\_total\_sim(M) \leq min(m_1, m_2) < max(m_1, m_2)$, resulting in a similarity value that is less than one. If either (i) or (ii) is satisfied, $SCD(a, b) \in [0, 1)$.

Thus, $SCD(a, b) \leq SCD(a, a)$

We have not been able to prove that Eq. (5.8) satisfies triangular inequality. In view of that, we shall refer to the formula in Eq. (5.8) as a similarity measure rather than a metric.

5.3 Summary

This chapter has discussed two methods of retrieving software projects from a repository based on the similarity of class diagrams. Shallow similarity assessment of class diagrams was formulated as a graph matching/similarity problem. Class diagrams were first represented as graphs. Next, nodes in the graphs were matched using either GA or CSA, in order to determine the degree of similarity of the graphs. Deep similarity assessment of class diagrams involved the comparison of constituent parts of class diagrams such as attribute lists, method lists and parameter lists.

Chapter 6

Functional Similarity Assessment

This chapter proposes techniques for retrieving software artifacts by comparing the functionality of new and existing software systems. Early in the development lifecycle, the functionality of software systems are manifested in the sequence diagrams which realize the different use cases of a system. Each use case is typically realized by one or more sequence diagrams showing how objects work together to carry out the use case [7].

The sequence diagrams similarity assessment problem comprises two subproblems; entity matching and similarity scoring. Entity matching entails mapping nodes in graph representation of sequence diagrams, classes appearing in sequence diagrams, or sequence diagrams from two collections of sequence diagrams. Determination of suitable mapping of entities may involve examination of large search spaces. Consequently, we use heuristic search techniques to find suitable mappings during entity matching. Similarity scoring is concerned with the similarity measures used to determine the degree of similarity of two sequence diagrams, or two sets of sequence diagrams. As a result of the very successful application of graph matching for determining the similarity between class diagrams in Chapter 5, we adopt a similar approach by computing the similarity score of two sequence diagrams from their graph representations. Furthermore, we formulate the similarity scoring of (sets of) sequence diagrams as a longest common

subsequence (LCS) problem. The LCS problem requires finding one of the longest sequences common to two strings by deleting zero or more symbols from the strings [59]. It has several applications such as in DNA or protein alignment, file comparison, gas chromatography and speech recognition [60]. This chapter presents an approach for determining sequence diagrams' similarity using the length of their longest common subsequence of matching messages (LCSMM). Two messages in different sequence diagrams match if the source and destination classes (or objects) of both messages are mapped.

The rest of the chapter is organized as follows. Section 6.1 presents a method of determining the similarity of two sequence diagrams from their graph representations. In Section 6.2, similarity assessment of two sequence diagrams from their LCSMM is discussed. The LCSMM-based method is extended to cater for the similarity of two sets of sequence diagrams in Section 6.3. A summary of the chapter is presented in Section 6.4.

6.1 Graph-Based Similarity Assessment of Two Sequence Diagrams

This section focuses on how to assess the similarity of sequence diagrams by comparing their graph representations. The approach is similar to that used to compare class diagrams in Chapter 5.

6.1.1 Graph Representation of Sequence Diagrams

Sequence diagrams can be converted to Message-Object-Order-Graphs (MOOGs), which are directed graphs in which there are nodes whenever messages are sent or received. In addition, there are message edges that denote the flow of messages between objects, and temporal edges that denote the flow of time within each object [37]. Figure 6.1 shows three sample sequence diagrams *a*, *b* and *c*, and the corresponding MOOG (*MOOG-a*, *MOOG-b*, *MOOG-c*) underneath each diagram. In the MOOGs, solid and dashed edges are the message and temporal edges, respectively. An adjacency matrix representation of *MOOG-a* is shown in Table 6.1. Rather than containing zeroes and ones, the entries of the matrix denote the types of edges of the graph.

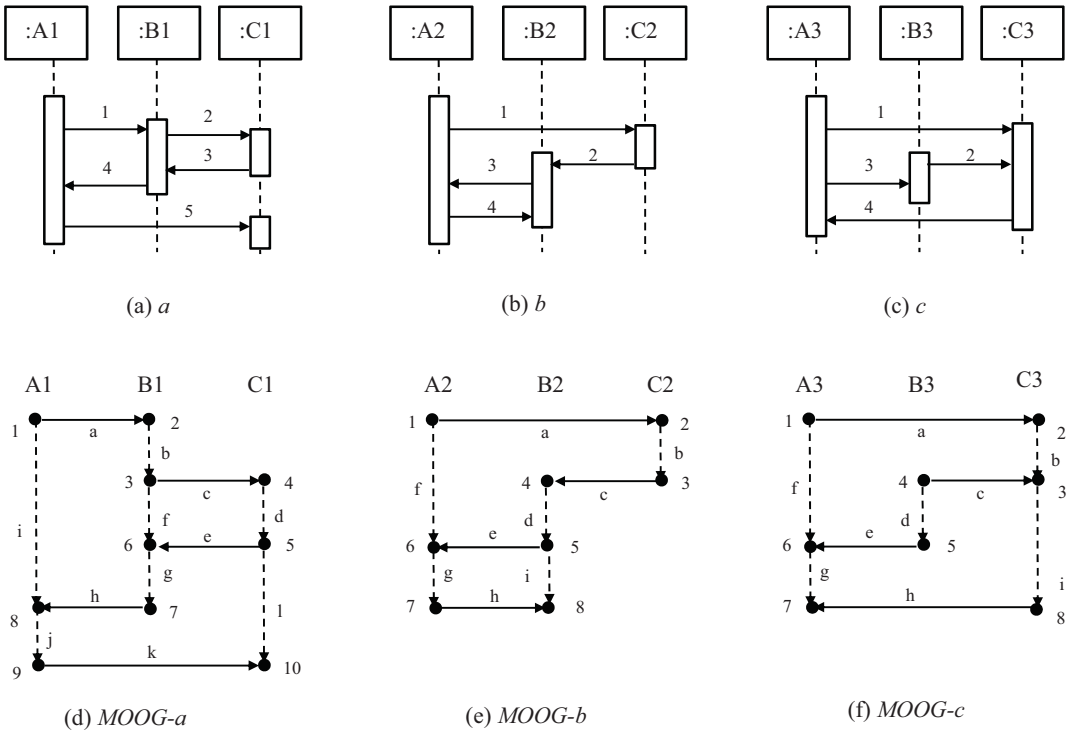


Figure 6.1: Three sequence diagrams *a*, *b* and *c*, and their corresponding message object order graphs [37]

6.1.2 Similarity Measure

In order to compare two sequence diagrams, the adjacency matrices of their MOOGs are compared. To facilitate the comparison, we define a mismatch matrix (*MM*), whose entries indicate whether or not two edges are of different types. The entries on the leading diagonal are zero, while the remaining entries are one. Table 6.2 shows *MM.*

Let *adj_a* and *adj_b* be the adjacency matrices of the *MOOG-a* and *MOOG-b*, respectively. *adj_a* has *na* rows while *adj_b* has *nb* rows ($na \leq nb$). Let *N* be a permutation vector that maps all *na* nodes of *adj_a* to *na* nodes of *adj_b*. Furthermore, let adj_b_N be a *na* X *na* adjacency matrix containing only the edges between nodes of *MOOG-b* listed in *N*. The degree of similarity between *a* and *b* is given in Eq. (6.1).

Table 6.1: Adjacency matrix of *MOOG-a* of Figure 6.1

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	-	M	-	-	-	-	-	T	-	-
2	-	-	T	-	-	-	-	-	-	-
3	-	-	-	M	-	T	-	-	-	-
4	-	-	-	-	T	-	-	-	-	-
5	-	-	-	-	-	M	-	-	-	T
6	-	-	-	-	-	-	T	-	-	-
7	-	-	-	-	-	-	-	M	-	-
8	-	-	-	-	-	-	-	-	T	-
9	-	-	-	-	-	-	-	-	-	M
10	-	-	-	-	-	-	-	-	-	-

- = No edge, T = Temporal edge, M = Message edge

Table 6.2: Mismatch matrix *MM*

	-	**T**	**M**
-	0	1	1
T	1	0	1
M	1	1	0

- = No edge, T = Temporal edge, M = Message edge

$$sim(a,b) = \frac{\sum_{i=1}^{na}\sum_{j=1}^{na} MM(adj\_a(i,j), adj\_b_N(i,j))}{nr} + \beta\frac{nb-na}{na} \quad (6.1)$$

where *nr* is the number of times there is at least one edge at corresponding entry positions in *adj_a* or *adj_b*$_N$. β ϵ [0, 1] is a weight that determines how the unmapped nodes in *MOOG-b* affect the degree of similarity. For example, choosing β = 0 causes the similarity score between *a* and *b* to be zero (indicating maximum similarity) whenever *b* subsumes *a*. On the other hand, large values of α causes the value of *sim*(*a, b*) to increase when *nb* > *na*.

In the following paragraphs, we prove that the similarity measure of Eq. (6.1) satisfies the first three metric axioms.

Self-similarity: Since the diagonal entries of *MM* are zero and corresponding edges of identical MOOGs are the same, the numerator of the first fraction in Eq. (6.1) is zero. Furthermore, identical MOOGs have the same number of nodes so the numerator of the second fraction in Eq. (6.1) is zero. Therefore, *sim*(*a*, *a*) = *sim*(*b*, *b*) = 0.

Minimality: There are two cases to consider:

Case 1: if *a* = *b*, it follows that *sim*(*a*, *b*) = *sim*(*a*, *a*) = 0 from the first axiom.

Case 2: if *a* ≠ *b*, either or both of the following conditions is true: (i) there is at least one pair of nodes whose corresponding edges differ in *MOOG-a* and *MOOG-b*. Since the non-diagonal entries of *MM* are one, the numerator of the first fraction in Eq. (6.1) is greater than zero (ii) *a* and *b* have different number of nodes, thus the numerator of the

second fraction in Eq. (6.1) is greater than zero. If condition (i) and/or (ii) is satisfied, $sim(a, b) \in (0, 1]$.

Thus, $sim(a, b) \geq sim(a, a)$

Symmetry: Clearly, $sim(a, b) = sim(b, a)$ since *MM* is symmetric.

We have not been able to prove that Eq. (6.1) satisfies triangular inequality, so we shall refer to the formula in Eq. (6.1) as a similarity measure rather than a metric.

6.1.3 Computation of Nodes' Similarity Matrix

This section describes a method of computing pairwise similarity between nodes of two MOOGs. The similarity values are kept in a nodes' similarity matrix *SM*, which will be used during matching. Each node is represented by a four-dimensional vector indicating four properties of a node: the number of message edges beginning at the node, the number of message edges ending at the node, the number of temporal edges beginning at the node and the number of temporal edges terminating at the node. Table 6.3 shows the features of nodes belonging to *MOOG-a* and *MOOG-b* of Figure 6.1. The similarity between two nodes is the Euclidean distance between their feature vectors. Table 6.4 shows matrix *SM* containing the pairwise similarities between nodes of *MOOG-a* and *MOOG-b*.

Table 6.3: Properties' matrix for *MOOG-a* and *MOOG-b* of Figure 6.1

	MOOG-a				***MOOG-b***			
Nodes	**# message edges going out**	**# message edges coming in**	**# temporal edges going out**	**# temporal edges coming in**	**# message edges going out**	**# message edges coming in**	**# temporal edges going out**	**# temporal edges coming in**
1	1	0	1	0	1	0	1	0
2	0	1	1	0	0	1	1	0
3	1	0	1	1	1	0	0	1
4	0	1	1	0	0	1	1	0
5	1	0	1	1	1	0	1	1
6	0	1	1	1	0	1	1	1
7	1	0	0	1	1	0	0	1
8	0	1	1	1	0	1	0	1
9	1	0	0	1				
10	0	1	0	1				

Table 6.4: Nodes' similarity matrix *SM*

		***MOOG-b* nodes**							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
***MOOG-a* nodes**	**1**	0.00	1.41	1.41	1.41	1.00	1.73	1.41	2.00
	2	1.41	0.00	2.00	0.00	1.73	1.00	2.00	1.41
	3	1.00	1.73	1.00	1.73	0.00	1.41	1.00	1.73
	4	1.41	0.00	2.00	0.00	1.73	1.00	2.00	1.41
	5	1.00	1.73	1.00	1.73	0.00	1.41	1.00	1.73
	6	1.73	1.00	1.73	1.00	1.41	0.00	1.73	1.00
	7	1.41	2.00	0.00	2.00	1.00	1.73	0.00	1.41
	8	1.73	1.00	1.73	1.00	1.41	0.00	1.73	1.00
	9	1.41	2.00	0.00	2.00	1.00	1.73	0.00	1.41
	10	2.00	1.41	1.41	1.41	1.73	1.00	1.41	0.00

6.1.4 Matching Using Heuristic Search Techniques

The heuristic search algorithms for determining an optimal permutation vector *N* when comparing two sequence diagrams are very similar to those described in Section 5.1.4

except for the following differences: each chromosome/nest is of the same form as *N*; the fitness of an individual is computed using Eq. (6.1); the fitness of a gene is read from the nodes' similarity matrix *SM*, which is also used during crossover and population initialization.

6.2 LCSMM-based Similarity Assessment of Two Sequence Diagrams

This section describes a method of determining the functional similarity of software systems from the length(s) of the LCSMM of their sequence diagrams. First, we describe how the LCSMM of two sequence diagrams is computed. Next, we present the similarity scoring formula for two sequence diagrams, and how matching is done with heuristic search techniques.

6.2.1 Longest Common Subsequence of Matching Messages (LCSMM)

The LCSMM we propose for sequence diagrams is analogous to the LCS of two strings. The LCS problem is to find one of the longest subsequences by deleting zero or more symbols from both strings [59]. For example, the longest common subsequence of TUeSDAY and ThUrSDAY is TUSDAY, which contains 6 symbols. The similarity measure of sequence diagrams is computed from the length of LCSMM rather than the LCSMM itself. In order to compute the length of LCSMM, the first step is to establish a mapping of classes from one sequence diagram to another. Thereafter, the length of LCSMM is computed according to the mapping. The messages in two sequence diagrams match if the source and receiving classes of the two messages are mapped.

Let *a* and *b* denote two sequence diagrams having |*a*| and |*b*| messages, respectively. Also, assume *a* involves *ca* classes a_1, a_2, … a_{ca}, while *b* involves *cb* classes b_1, b_2, … b_{cb} ($ca \leq cb$). Furthermore, let, $a_{i,\,src}$ and $a_{i,\,dest}$ denote the source and destination classes for the i^{th} message of *a* respectively, and analogously for *b*. For example, $a_{2,\,\mathrm{src}} = 1$ and $a_{2,\,\mathrm{dest}} = 3$ means that the second message of *a* is sent from an object of a_1 to that of a_3. Let *C* be a permutation vector mapping all *ca* classes of *a* to *ca* classes of *b*. Figure 6.2 shows how *C* translates to a mapping of classes.

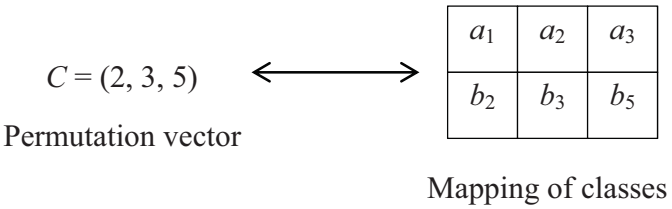


Figure 6.2: Mapping of classes using a permutation vector

The length of the LCSMM of *a* and *b*, given a permutation vector *C* is denoted by $L_C(a, b)$. Its value can be obtained from $LCS_C(|a|, |b|)$. The latter value is computed using a dynamic programming algorithm having computational complexity of O(|*a*|*|*b*|) by applying the following recursive formula, which was adapted from [61]:

$$LCS_C(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS_C(i-1, j-1) + 1 & \text{if } i > 0, j > 0 \text{ and } C(a_{i,src}) = b_{j,src} \\ & \text{and } C(a_{i,dest}) = b_{j,dest} \\ \max\{LCS_C(i-1, j), LCS_C(i, j-1)\} & \text{otherwise} \end{cases}$$

where *max* is a function that returns the larger of its two arguments. Figure 6.3 shows two sample sequence diagrams *a* and *b*. Only the matching messages in both sequence diagrams are labelled in Figure 6.4. The value of $L_C(a, b)$ using $C = (1, 3, 4)$ is 5.

Intermediate and final values of LCS_C are shown in Table 6.5, where row and column headings are the message sequence numbers.

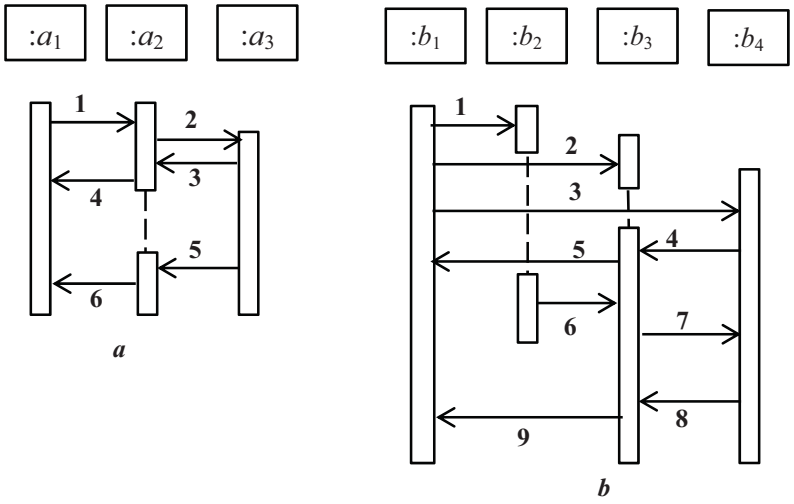


Figure 6.3: Two sample sequence diagrams *a* and *b*

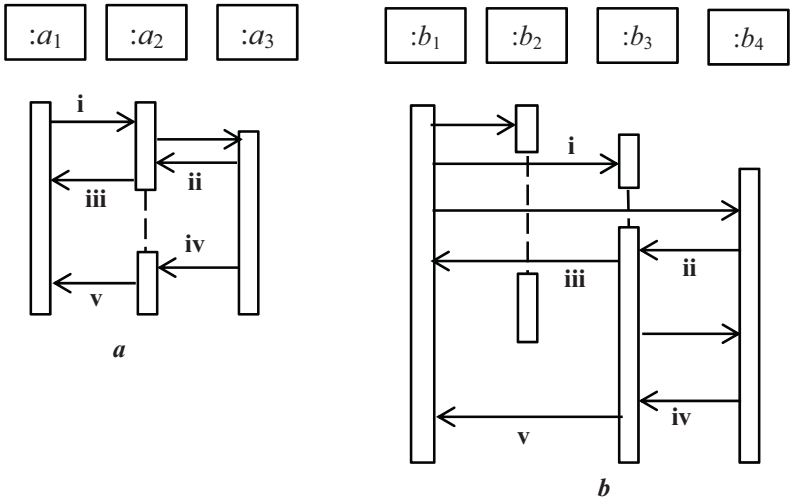


Figure 6.4: Mapped messages forming LCSMM

Table 6.5: Values of LCS_C used in computing L_C

		message numbers for *b*									
		-	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
message numbers for *a*	**-**	0	0	0	0	0	0	0	0	0	0
	1	0	0	1	1	1	1	1	1	1	1
	2	0	0	1	1	1	1	1	2	2	2
	3	0	0	1	1	2	2	2	2	3	3
	4	0	0	1	1	2	3	3	3	3	4
	5	0	0	1	1	2	3	3	3	4	4
	6	0	0	1	1	2	3	3	3	4	5

6.2.2 Similarity Score of Two Sequence Diagrams

Intuitively, two similar sequence diagrams should have many matching messages. Thus, we use the value of L_C as an indicator of the degree of similarity of two sequence diagrams. The degree of similarity of *a* and *b* is defined in Eq. (6.2).

$$s(a,b) = 1 - \frac{2 * L_C(a,b)}{|a| + |b|} \tag{6.2}$$

We describe the use of GA and CSA to obtain a suitable class permutation vector *C* in Section 6.2.3. The possible values of *s* lie in the range [0, 1]. A similarity value of 0 indicates maximum degree of similarity between two sequence diagrams, while a value of 1 represents the least possible degree of similarity. The optimal value of *C* for the two sequence diagrams presented in Figure 6.3 is *C* = (1, 3, 4). This value was obtained experimentally with the aid of GA. Using Eq. (6.2), the similarity score of the two sequence diagrams is:

$s(a, b) = 1 - 2*5/(6+9) = 0.33.$

In the following paragraphs, we prove that the similarity measure of Eq. (6.2) satisfies the first three metric axioms.

Self-similarity: Since $L_C(a, a) = |a|$ and $L_C(b, b) = |b|$, it follows that $s(a, a) = s(b, b) = 0$.

Minimality: There are two cases to consider:

Case 1: if $a = b$, it follows that $s(a, b) = s(a, a)$.

Case 2: if $a \neq b$, either or of the following conditions is true: (i) $|a| \neq |b|$ so $L_C(a, b)$ is at most as large as the minimum of $|a|$ and $|b|$. Consequently, $2*L_C(a, b) < (|a| + |b|)$ and the fractional part of Eq. (6.2) is strictly less than 1. (ii)) $|a| = |b|$ but since $a \neq b$, $L_C(a, b) < |a|$. Again, this results in the fractional part of Eq. (6.2) being strictly less than 1. If condition (i) or (ii) is satisfied, $s(a, b) \in (0, 1]$.

Thus, $s(a, b) \geq s(a, a)$

Symmetry: $s(a, b) = s(b, a)$ only if $L_C(a, b) = L_C(b, a)$. Like the length of LCS of two strings, the length of LCSMM is symmetric by definition. In other words, $L_C(a, b) = L_C(b, a)$ and thus $s(a, b) = s(b, a)$.

We have been unable to prove that Eq. (6.2) satisfies triangular inequality, so we shall refer to the formula in Eq. (6.2) as a similarity measure rather than a metric.

6.2.3 Determining the Permutation Vector for Sequence Diagrams

The heuristic search algorithms for determining C when comparing two sequence diagrams are similar to those described in Section 5.1.4, except for a few differences: each chromosome/nest is of the same form as C; the fitness of an individual is computed

using Eq. (6.2); a different classes' similarity matrix is used during population initialization and crossover, as well as for determining the fitness of a gene. This section describes how the similarity matrix is computed.

The similarity matrix contains pairwise similarity values between classes in the two sequence diagrams to be compared. Each class is represented by a feature vector indicating two properties of the class; the number of messages received and the number of messages sent. A classes' properties matrix is created, such that each row holds the properties for one class. The property matrix for the two sequence diagrams of Figure 6.3 is shown in Table 6.6. The Euclidean distance measure is used to compute the similarity between two classes. Table 6.7 shows the pairwise similarities between classes in *a* and *b*.

Table 6.6: Properties' matrix for the classes in *a* and *b* of Figure 6.3

		number of messages sent	**number of messages received**
a	**Class 1**	1	2
	Class 2	3	3
	Class 3	2	1
b	**Class 1**	3	2
	Class 2	1	1
	Class 3	3	4
	Class 4	2	2

Table 6.7: Classes' similarity matrix of *a* and *b* of Figure 6.3

		b			
		Class 1	**Class 2**	**Class 3**	**Class 4**
a	**Class 1**	2.00	1.00	2.83	1.00
	Class 2	1.00	2.83	1.00	1.41
	Class 3	1.41	1.00	3.16	1.00

6.3 LCSMM-based Similarity Assessment for Sets of Sequence Diagrams

Software systems are rarely modeled using a single sequence diagram. Rather, the functionality of software systems are represented by means of use case diagrams. Each use case is then typically realized using one or more sequence diagrams. Thus, this section extends the similarity assessment technique of Section 6.2 to cater for the comparison of two sets of sequence diagrams.

6.3.1 Similarity Score of Two Sets of Sequence Diagrams

Let A denote a set of SA sequence diagrams (A_1, $A_2 \ldots A_{SA}$). These sequence diagrams collectively involve CA classes a_1, $a_2 \ldots$ a_{CA}. Similarly, let B be a set of SB sequence diagrams (B_1, $B_2 \ldots B_{SB}$) collectively involving CB classes b_1, $b_2 \ldots$ b_{CB}. The number of messages in A_i is denoted as $|A_i|$ ($1 \leq i \leq SA$), and similarly for B_i ($1 \leq i \leq SB$). The degree of similarity between A and B is given by Eq. (6.3).

$$sim(A,B) = \begin{cases} 1 - \frac{1}{SA}\sum_{i=1}^{SA} \frac{2 * L_C(A_i, B_{S(i)})}{|A_i| + |B_{S(i)}|} + 2\gamma \frac{SB - SA}{SA + SB} & \text{if } CA \leq CB, SA \leq SB \\ 1 - \frac{1}{SB}\sum_{i=1}^{SB} \frac{2 * L_C(A_{S(i)}, B_i)}{|A_{S(i)}| + |B_i|} + 2\gamma \frac{SA - SB}{SA + SB} & \text{if } CA \leq CB, SA > SB \\ 1 - \frac{1}{SA}\sum_{i=1}^{SA} \frac{2 * L_C(B_{S(i)}, A_i)}{|B_{S(i)}| + |A_i|} + 2\gamma \frac{SB - SA}{SA + SB} & \text{if } CA > CB, SA \leq SB \\ 1 - \frac{1}{SB}\sum_{i=1}^{SB} \frac{2 * L_C(B_i, A_{S(i)})}{|B_i| + |A_{S(i)}|} + 2\gamma \frac{SA - SB}{SA + SB} & \text{if } CA > CB, SA > SB \end{cases} \quad (6.3)$$

C and S are suitable permutation vectors for mapping classes and sequence diagrams, respectively in A and B. A constant, $\gamma \in [0, 1]$ determines how unmatched sequence

diagrams are handled. When there are unmatched sequence diagrams, large values of γ increase the value of *sim*(*A, B*) indicating a smaller degree of similarity between *A* and *B*.

The following example illustrates how Eq. (6.3) is used to compute the similarity score between *A* and *B* given in Figure 6.5. We use $C = (1, 2, 3)$, $S = (2, 1)$ and $\gamma = 0.15$. These values were determined experimentally.

From Figure 6.5, $CA = 3$, $SA = 2$, $CB = 4$, $SB = 2$, $|A_1| = |A_2| = |B_1| = |B_2| = 4$.

The first form of Eq. (6.3) is used since $CA \leq CB$ and $SA \leq SB$.

$$sim(A,B) = 1 - \frac{1}{2}\left(\frac{2*\mathrm{L_C}(A_1,B_2)}{|A_1|+|B_2|} + \frac{2*\mathrm{L_C}(A_2,B_1)}{|A_2|+|B_1|}\right) + 2\gamma\frac{SB-SA}{SA+SB}$$

$$= 1 - \frac{1}{2}\left(\frac{6}{8}+\frac{6}{8}\right) + 0.3\left(\frac{2-2}{2+2}\right) = 0.25$$

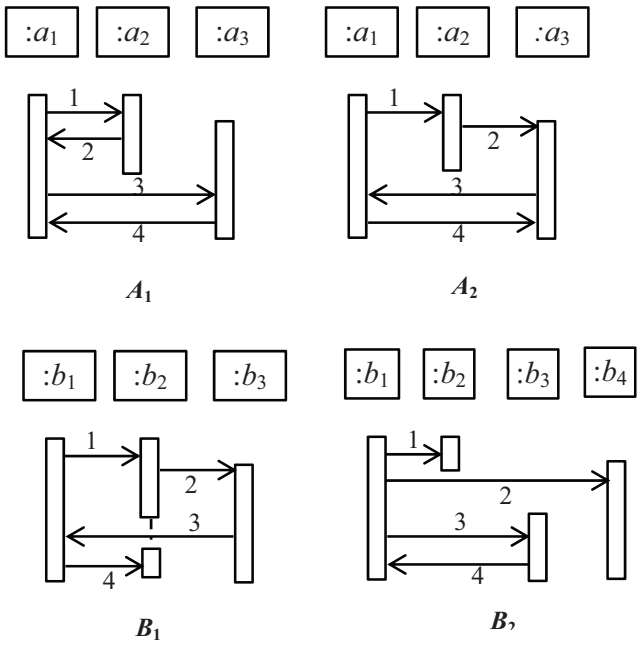


Figure 6.5: Two sample sets of sequence diagrams $A = (A_1, A_2)$ and $B = (B_1, B_2)$

We shall now prove that the similarity measure of Eq. (6.3) satisfies the first three metric axioms.

Self-similarity: Because all the corresponding messages of identical sets of sequence diagrams match, the first fraction in Eq. (6.3) evaluates to one. Furthermore, identical sets of sequence diagrams contain equal number of sequence diagrams so the numerator of the second fraction in Eq. (6.3) is zero. Therefore, $sim(A, A) = sim(B, B) = 0$.

Minimality: There are two cases to consider:

Case 1: if $A = B$, it follows that $sim(A, B) = sim(A, A) = 0$ from the first axiom.

Case 2: if $A \neq B$, either or both of the following conditions is true: (i) there is at least one pair of sequence diagrams whose corresponding messages do not exactly match. As a result, the first fraction in Eq. (6.3) evaluates to less than one. (ii) A and B have different number of sequence diagrams, thus the numerator of the second fraction in Eq. (6.3) is greater than zero. If (i) and/or (ii) holds, $sim(A, B) \in (0, 1]$.

Thus, $sim(A, B) \geq sim(A, A)$

Symmetry: $sim(A, B) = sim(B, A)$ because, as stated in Section 6.2.2, $L_C(\ldots)$ is a symmetric function.

We have not been able to prove that Eq. (6.3) satisfies triangular inequality. We shall therefore refer to the formula in Eq. (6.3) as a similarity measure rather than a metric.

6.3.2 Computation of Similarity Matrices for Sets of Sequence Diagrams

Two similarity matrices are required for comparing sets of sequence diagrams: a classes' similarity matrix and a sequence diagrams' similarity matrix. Each class is represented by a feature vector indicating four properties of the class; the total number of messages sent, number of sequence diagrams in which the class sends messages, total number of messages received and number of sequence diagrams in which the class receives messages. Each sequence diagram is represented by a feature vector indicating four properties of the sequence diagram; the number of classes in the diagram, number of messages in the diagram, standard deviation of number of messages sent by classes and standard deviation of number of messages received by classes. The property matrices for classes and sequence diagrams of Figure 6.5 are shown in Table 6.8 and Table 6.9, respectively. Again, the Euclidean distance measure is used to compute the pairwise similarities between classes (see Table 6.10) and pairwise similarities between sequence diagrams (see

Table 6.11).

Table 6.8: Classes' properties matrix for diagrams in Figure 6.5

Sequence Diagram Set	Class	# messages sent	# sequence diagrams in which message was sent	# messages received	# sequence diagrams in which message was received
A	**Class 1**	4	2	3	2
	Class 2	2	2	2	2
	Class 3	2	2	3	2
B	**Class 1**	5	2	2	2
	Class 2	1	1	3	2
	Class 3	2	2	2	2
	Class 4	0	0	1	1

Table 6.9: Sequence diagrams' properties matrix for diagrams in Figure 6.5

	# messages	# classes	Standard deviation of # messages sent by classes	Standard deviation of # messages received by classes
A_1	4	3	0.47	0.47
A_2	4	3	0.47	0.47
B_1	4	3	0.47	0.47
B_2	4	4	1.00	1.00

Table 6.10: Classes' similarity matrix for diagrams in Figure 6.5

		B			
		Class 1	**Class 2**	**Class 3**	**Class 4**
A	**Class 1**	1.41	3.16	2.24	5.00
	Class 2	3.00	1.73	0.00	3.16
	Class 3	3.16	1.41	1.00	3.61

Table 6.11: Sequence diagrams' similarity matrix for diagrams in Figure 6.5

	Diagram B_1	**Diagram B_2**
Diagram A_1	0.00	1.23
Diagram A_2	0.00	1.23

6.3.3 Determination of Permutation Vectors Using GA

This section briefly describes the use of GA to find suitable permutation vectors *C* and *S* without exhaustively searching through all possible values. Because the GA works in a similar manner as previously described in Section 5.1.4, only the differences are mentioned.

Each chromosome is made up of two parts. The first part handles the mapping of classes in the sets of sequence diagrams *A* and *B*, while the second part of the chromosome maps the sequence diagrams in *A* and *B*. Let *minC* and *maxC* be the smaller and larger of the

values *CA* and *CB*, respectively. Likewise, let *minS* and *maxS* be the smaller and larger of the values *SA* and *SB*, respectively. A suitable encoding of a chromosome to determine a mapping of classes and sequence diagrams between *A* and *B* is shown in Figure 6.6. For example, the second, fifth and $maxC^{th}$ classes of the set of diagrams having more classes is mapped to the first, second and third classes of the set of diagrams having fewer classes. The mapping of sequence diagrams can be inferred similarly from the second part of the chromosome. Thus, a chromosome can be thought of as a concatenation of *C* and *S*. Figure 6.7 shows a possible chromosome encoding for computing the similarity score of the two sets of sequence diagrams in Figure 6.5.

At the beginning of GA a single individual may be constructed by applying Munkres algorithm on the classes' similarity matrix (to obtain an initial mapping of classes) and sequence diagrams' similarity matrix (to obtain an initial mapping of sequence diagrams). The first individual is formed by concatenating the two assignment vectors returned by Munkres algorithm. A few additional individuals are subsequently generated by replicating the first individual and mutating it. All other individuals in the population are generated randomly. Crossover, mutation and selection operations work in the same way as described in Section 5.1.4. The fitness value of an individual is computed using Eq. (6.3). The fitness of a gene is read off of the classes' similarity matrix or the sequence diagrams' similarity matrix depending on which part of the chromosome the gene is located.

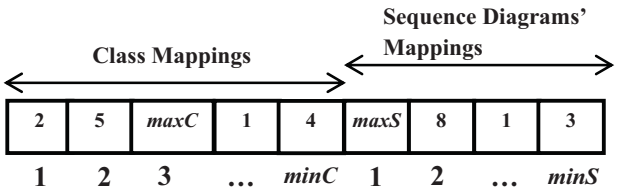


Figure 6.6: Chromosome encoding for computing similarity of two sets of sequence diagrams

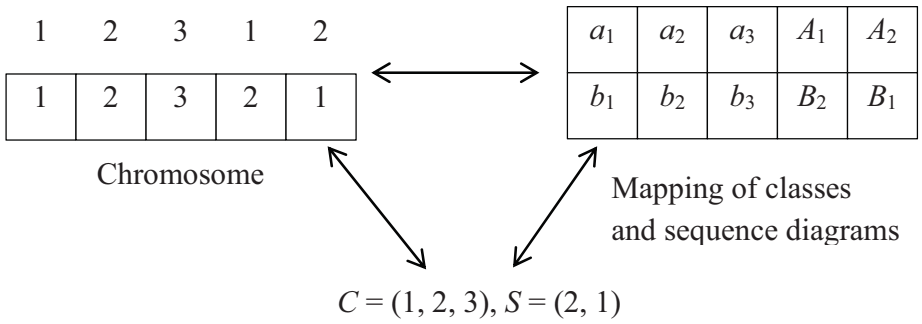


Figure 6.7: Possible chromosome encoding for comparing the diagrams of Figure 6.5

6.4 Summary

This chapter has described two manners of computing the similarity of two sequence diagrams: by comparing their graph representations; and from the length of their longest common subsequence of matching messages.

In practice, functionality of software systems are commonly represented using sets of sequence diagrams rather than single sequence diagrams. Thus, the method of comparing two sequence diagrams based on their sequence of matching messages was extended to cater for sets of sequence diagrams.

Chapter 7

Behavioral Similarity Assessment

UML state machine diagrams depict the behavior of an object in a system by showing how events lead to changes in the state of the object during its life time. This chapter describes how the similarity between two state machine diagrams can be determined from their graph representations. In addition, the similarity assessment of groups of state machine diagrams is discussed.

Section 7.1 proposes a graphical representation of state machine diagrams. The similarity measure for state machine diagrams is presented in Section 7.2. Computation of states' similarity matrix is discussed in Section 7.3. This matrix is utilized during matching, which is covered in Section 7.4. Similarity assessment of groups of state machine diagrams is discussed in Section 7.5, while Section 7.6 summarizes the chapter.

7.1 Graph Representation of State Machine Diagrams

State machine diagrams can be converted to labeled directed graphs in which every state other than a final state is represented by a node, and all final states are represented by a single node. There are four types of edges connecting the nodes: hierarchical edges labelled *H*, which connect composite states to their immediate sub states; transition edges labelled *xT*, which represent transitions between states, where *x* is the number of

transitions from one state to another; beginning edges labelled *B*, which denote transitions from the start state; and ending edges labelled *xE*, which represent transitions to the end state, where *x* is the number of transitions from one state to any of the final states. Figure 7.1 shows two state machine diagrams *s* and *t*. The graph and adjacency matrix representations of *s* are shown in Figure 7.2 and Table 7.1, respectively.

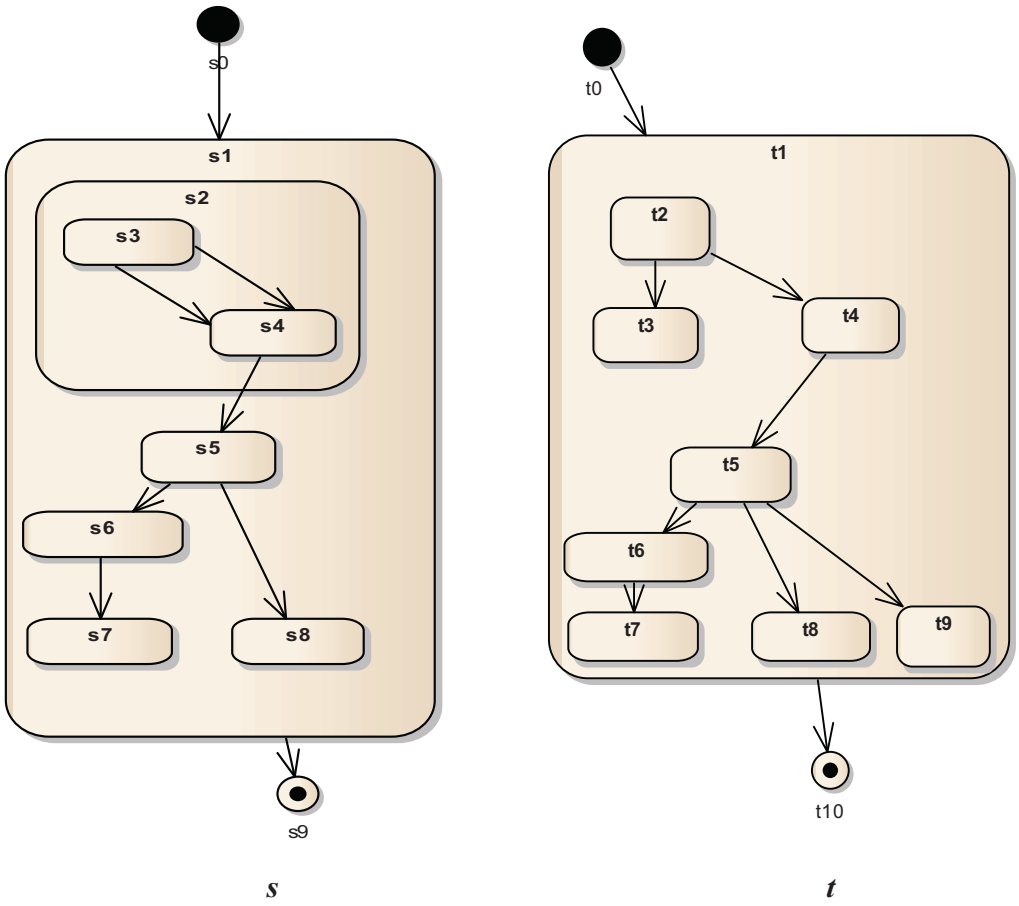


Figure 7.1: Two state machine diagrams *s* and *t* (adapted from [62])

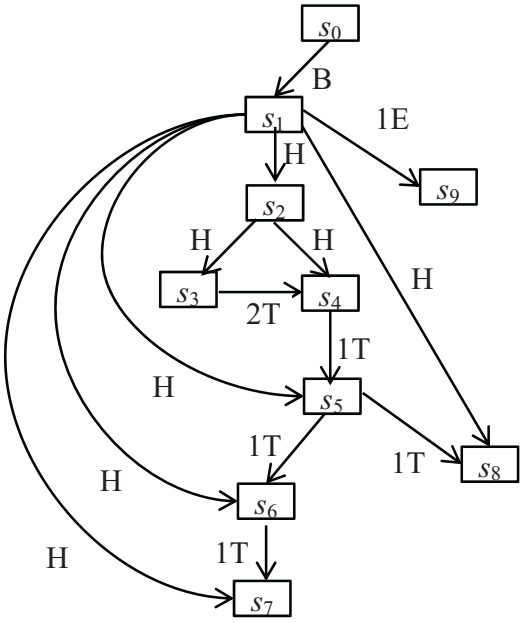


Figure 7.2: Graph representation of *s*

Table 7.1: Adjacency matrix representation of *s*

	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9
s_0	-	B	-	-	-	-	-	-	-	-
s_1	-	-	H	-	-	H	H	H	H	1E
s_2	-	-	-	H	H	-	-	-	-	-
s_3	-	-	-	-	2T	-	-	-	-	-
s_4	-	-	-	-	-	1T	-	-	-	-
s_5	-	-	-	-	-	-	1T	-	1T	-
s_6	-	-	-	-	-	-	-	1T	-	-
s_7	-	-	-	-	-	-	-	-	-	-
s_8	-	-	-	-	-	-	-	-	-	-
s_9	-	-	-	-	-	-	-	-	-	-

B = beginning edge, *H* = hierarchical edge, *xT* = *x* transition edges, *yE* = *y* ending edges, - = no edge

7.2 Similarity Measure for Two State Machine Diagrams

The degree of similarity of two state machine diagrams is computed by comparing their adjacency matrix representations. A difference matrix *DiffS* acts as a lookup table that indicates the degree of similarity between the four different types of edges. Table 7.2

shows *DiffS*. The non-diagonal entries of *DiffS* are ones, indicating maximum dissimilarity. The diagonal entries for beginning edges and hierarchical edges are zero, signifying that identical types of edges have no difference between them. However, in the case of transition edges and ending edges, their labels indicate the number of transitions, hence the diagonal entries of *DiffS* take these numbers into account. For example, the difference between a 2*T* edge and a 3*T* edge is 1/2 – 1/3 = 0.17, whereas the difference between a 2*T* edge and a 4*T* edge is 1/2 – 1/4 = 0.25.

Table 7.2: *DiffS*

	B	***H***	***yT***	***yE***
B	0	1	1	1
H	1	0	1	1
xT	1	1	$\lvert 1/x - 1/y \rvert$	1
yE	1	1	1	$\lvert 1/x - 1/y \rvert$

B = beginning edge, *H* = hierarchical edge, *xT* or *yT* = *x* or *y* transition edges, *xE* or *yE* = *x* or *y* ending edges, - = no edge, |…| = absolute value

Let *adj_s* and *adj_t* be the adjacency matrices of *s* and *t*, respectively. *adj_s* has *ns* rows while *adj_t* has *nt* rows ($ns \leq nt$). Let *K* be a permutation vector that maps all *ns* nodes of *adj_s* to *nt* nodes of *adj_t*. Furthermore, let adj_t_K be a *ns* X *ns* adjacency matrix containing only the edges between nodes of *adj_t* listed in *K*. The degree of similarity between *s* and *t* is given in Eq. (7.1):

$$sim(s,t) = \frac{\sum_{i=1}^{ns} \sum_{j=1}^{ns} DiffS(adj\_s(i,j), adj\_t_K(i,j))}{nr} + \lambda \frac{nt - ns}{ns} \tag{7.1}$$

where *nr* is the number of times there is at least one edge at corresponding entry positions in *adj_s* or adj_t_K. $\lambda \in [0, 1]$ is a weight that determines how the unmapped nodes in *adj_t* affect the degree of similarity. For example, choosing $\lambda = 0$ causes the similarity score

between s and t to be zero (indicating maximum similarity) whenever t subsumes s. On the other hand, large values of λ cause the value of $sim(s, t)$ to increase when $nt > ns$.

In the sequel, we prove that the similarity measure of Eq. (7.1) satisfies the first three metric axioms.

Self-similarity: Since corresponding edges of identical state machine diagrams are the same, and the diagonal entries of *DiffS* are either zero or reflect the differences in number of edges, the numerator of the first fraction in Eq. (7.1) is zero. Furthermore, graph representations of identical state machine diagrams have the same number of nodes so the numerator of the second fraction in Eq. (7.1) is zero. Therefore, $sim(s, s) = sim(t, t) = 0$.

Minimality: There are two cases to consider:

Case 1: if $s = t$, it follows that $sim(s, t) = sim(s, s) = 0$ from the first axiom.

Case 2: if $s \neq t$, either or both of the following conditions is true: (i) there is at least one pair of nodes whose corresponding edges in adj_s and adj_t_K are of different types or have different multiplicities. Thus, the numerator of the first fraction in Eq. (7.1) is greater than zero (ii) s and t have different number of nodes, thus the numerator of the second fraction in Eq. (7.1) is greater than zero. If condition (i) and/or (ii) is satisfied, $sim(s, t) \in (0, 1]$.

Thus, $sim(s, t) \geq sim(s, s)$

Symmetry: Clearly, $sim(s, t) = sim(t, s)$ since *DiffS* is symmetric.

We have not been able to prove that Eq. (7.1) satisfies triangular inequality. Thus, we shall refer to the formula in Eq. (7.1) as a similarity measure rather than a metric.

7.3 Computation of States' Similarity Matrix

We now describe a method of computing pairwise similarities between states of two state machine diagrams. The similarity values are kept in a states' similarity matrix *SS*, which will be used during matching. Each state (apart from the final states which are listed as one state) is represented by a 10-dimensional vector indicating 10 properties of the state. These properties are listed in Table 7.3, while their values are given in Table 7.4 for *s* and *t*. The similarity between nodes is the Euclidean distance between their feature vectors. Table 7.5 shows *SS* containing the pairwise similarities between states in *s* and *t*.

Table 7.3: Features of each state

Feature	Description
f_1	Number of transitions coming from the start state
f_2	Number of transitions coming in (except from the start state)
f_3	Number of transitions to a finish state
f_4	Number of transitions going out (except to finish states)
f_5	Number of states whose next state is this state
f_6	Number of next states
f_7	Number of ancestors
f_8	Number of descendants
f_9	Number of child states
f_{10}	Length of longest path from this state to its descendants

7.4 Matching Using Heuristic Search Techniques

The heuristic search algorithms for determining permutation vector *K* when comparing two state diagrams are very similar to those described in Section 5.1.4 except for the following differences: each chromosome (or nest) is of the same form as *K*; the fitness of an individual is computed using Eq. (7.1); the fitness of a gene is read from the states' similarity matrix *SS*, which is also used during crossover and population initialization.

Table 7.4: Features of *s* and *t*

	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
s_0	0	0	0	1	0	1	0	0	0	0
s_1	1	0	1	0	1	1	0	7	5	2
s_2	0	0	0	0	0	0	1	2	2	1
s_3	0	0	0	2	0	0	2	0	0	0
s_4	0	2	0	1	0	1	2	0	0	0
s_5	0	1	0	2	1	2	1	0	0	0
s_6	0	1	0	1	1	1	1	0	0	0
s_7	0	1	0	0	1	0	1	0	0	0
s_8	0	1	0	0	1	0	1	0	0	0
s_9	0	1	0	0	1	0	0	0	0	0
t_0	0	0	0	1	0	1	0	0	0	0
t_1	1	0	1	0	1	1	0	8	8	1
t_2	0	0	0	2	0	2	1	0	0	0
t_3	0	1	0	0	1	0	1	0	0	0
t_4	0	1	0	1	1	1	1	0	0	0
t_5	0	1	0	3	1	3	1	0	0	0
t_6	0	1	0	1	1	1	1	0	0	0
t_7	0	1	0	0	1	0	1	0	0	0
t_8	0	1	0	0	1	0	1	0	0	0
t_9	0	1	0	0	1	0	1	0	0	0
t_{10}	0	1	0	0	1	0	0	0	0	0

Table 7.5: States' similarity matrix *SS*

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
s_0	0.00	11.53	1.73	2.24	1.73	3.32	1.73	2.24	2.24	2.24	2.00
s_1	9.06	3.32	9.33	9.11	9.11	9.75	9.11	9.11	9.11	9.11	9.06
s_2	3.46	8.77	4.12	3.32	3.61	5.39	3.61	3.32	3.32	3.32	3.46
s_3	2.45	11.87	2.24	2.65	2.24	3.61	2.24	2.65	2.65	2.65	3.16
s_4	2.83	11.87	2.65	2.24	1.73	3.32	1.73	2.24	2.24	2.24	2.83
s_5	2.24	11.75	1.41	2.83	1.41	1.41	1.41	2.83	2.83	2.83	3.00
s_6	1.73	11.58	2.00	1.41	0.00	2.83	0.00	1.41	1.41	1.41	1.73
s_7	2.24	11.58	3.16	0.00	1.41	4.24	1.41	0.00	0.00	0.00	1.00
s_8	2.24	11.58	3.16	0.00	1.41	4.24	1.41	0.00	0.00	0.00	1.00
s_9	2.00	11.53	3.32	1.00	1.73	4.36	1.73	1.00	1.00	1.00	0.00

7.5 Similarity Assessment of Groups of State Machine Diagrams

This section describes how to compute the similarity score of two sets of state machine diagrams. Let $S = S_1, S_2 \ldots S_{NS}$ and $T = T_1, T_2 \ldots T_{NT}$ be two groups of state machine diagrams for two software systems ($NS \leq NT$). The degree of similarity between S and T can be computed in four steps:

(i) Form a matrix of pairwise similarities between state machine diagrams in S and T.

(ii) Use Munkres' algorithm to find the *minimum total similarity* between S and T.

(iii) Divide the *minimum total similarity* by NS to obtain *minimum average similarity*

(iv) Increase the similarity value if $NS \neq NT$

Eq. (7.2) gives the degree of similarity between S and T:

$$sim(S,T) = \frac{\text{minimum total similarity}(S,T)}{NS} + \theta \frac{NT - NS}{NS} \qquad (7.2)$$

Θ: ϵ [0, 1] determines how unmatched state machine diagrams are handled. When there are unmatched state machine diagrams (i.e., when $NS < NT$), large values of Θ increase the value of $sim(S, T)$ indicating a smaller degree of similarity between S and T.

We shall now prove that the similarity measure of Eq. (7.2) satisfies the first three metric axioms.

Self-similarity: If two sets of state machine diagrams are identical, the numerator of the first fraction in Eq. (7.2) is computed by applying Munkres' algorithm on a square

matrix, say *M*. There is at least one entry in each row and each column of *M* that contains 0, indicating optimum similarity between two state machine diagrams. Therefore, the value returned by Munkres' algorithm is zero. Furthermore, identical sets of state machine diagrams contain equal number of state machine diagrams so the numerator of the second fraction in Eq. (7.2) is zero. Thus, Eq. (7.2) returns an optimal similarity value of zero for any two identical groups of state machine diagrams.

Minimality: There are two cases to consider:

Case 1: if $S = T$, it follows that $sim(S, T) = sim(S, S) = 0$ from the axiom of self-similarity.

Case 2: if $S \neq T$, assume the numerator of Eq. (7.2) is *minimum total similarity*(*M*) / *NS* where *M* is a *NS* x *NT* matrix of pairwise similarity values. Either or both of the following conditions is true: (i) there is at least one state machine diagram which cannot be mapped to an identical state machine diagram in the other class diagram (i.e., at least one row or one column does not contain a value of zero), thus *minimum total similarity* (*M*) > 0, resulting in a similarity value that is greater than zero. (ii) *S* and *T* have different number of state machine diagrams i.e., $NS \neq NT$, thus the numerator of the second fraction in Eq. (7.2) is greater than zero. If (i) and/or (ii) is satisfied, $sim(S, T) \in (0, 1]$.

Thus, $sim(S, T) \geq sim(S, S)$

Symmetry: $sim(S, T) = sim(T, S)$ because applying Munkres' algorithm on a matrix *M* gives the same optimal similarity as applying the algorithm on the transpose of *M*.

We have not been able to prove that Eq. (7.2) satisfies triangular inequality. We shall therefore refer to the formula in Eq. (7.2) as a similarity measure rather than a metric.

7.6 Summary

This chapter discussed the similarity assessment of state machine diagrams. A method of converting state machine diagrams to directed graphs was proposed. The directed graphs were compared by means of a similarity measure and GA.

Since state machine diagrams model the behavior of individual objects (i.e. class instances) in a system, it is not uncommon that several state machine diagrams are used to model a software system. Thus, we also described how to handle the similarity assessment of groups of state machine diagrams.

Chapter 8

Multi-view Similarity Assessment

Software is commonly modelled using several UML diagrams that show a system from different viewpoints rather than from only one perspective. This chapter discusses a number of ways of computing an overall similarity score of software systems by combining information from multiple viewpoints. The approaches for multi-view similarity assessment take into account the issues raised in Section 2.2. In each approach, multi-view similarity scores are computed by aggregating the similarity scores from individual views.

Consistent mapping of classes in class diagrams and state machine diagrams (I_2) can be easily handled. State machine diagrams show how an object responds to events according to its current state, and how it enters into new states [16]. Thus, class mappings in class diagrams implicitly determine the mapping of state machine diagrams that portray the behavior of objects of the mapped classes. In other words, once a pair of classes has been mapped, the state machine diagrams depicting the behavior of objects of both classes can be compared using a similarity scoring algorithm [63].

Systematic mapping of sequence diagrams in two groups of sequence diagrams (I_3) was addressed during functional similarity assessment. As can be seen from Figure 6.6, the

second part of the chromosome handles the mapping of multiple sequence diagrams in two requirement specifications.

The only issue that has not been addressed so far is the consistent mapping of classes in class diagrams and sequence diagrams (I_1). Recall that shallow structural similarity assessment (see Section 5.1) uses a permutation vector *P* to map classifiers in two class diagrams. Furthermore, functional similarity assessment (see Section 6.3) is carried out using a classes' permutation vector *C* and a sequence diagrams' permutation vector *S*. Sections 8.1, 8.2 and 8.3 describe three ways of managing the consistency between *P* and *C*.

8.1 View Composition Approach

Similarity computation via composition means that the multi-view similarity score is calculated as a weighted sum of independently computed similarity values for each view. As shown in Figure 8.1, the overall similarity score is then scaled by a factor – an *inconsistency penalty* - which takes into account the (possibly) conflicting mapping of entities in the different views [5]. *P* and *sim_structural* are the classifiers' permutation vector and structural similarity score respectively, for the fittest individual in the final population. Likewise, *C* and *sim_functional* are the classes' permutation vector and functional similarity score, respectively for the best individual at the end of GA. The formula for computing multi-view similarity score using composition approach is given in Eq. (8.1).

$$sim\_multi = (w_1 * sim\_structural + w_2 * sim\_functional + w_3 * sim\_behavioral) * (1 + ipenalty) \quad (8.1)$$

sim_structural, *sim_functional* and *sim_behavioral* are computed using Eq. (5.1), Eq. (6.3) and Eq. (7.2), respectively. w_1, w_2 and w_3 are constants that determine the relative weights of structural, functional and behavioral similarity scores, respectively. Note that w_1, w_2 and $w_3 \in [0, 1]$ and $w_1 + w_2 + w_3 = 1$. The inconsistency penalty, *ipenalty* can be computed as:

ipenalty = (number of mismatches in *P* and *C*)/(size of *P*).

Thus, if *P* and *C* are the same (i.e., there are no inconsistencies), *ipenalty* is zero. However, assume that *P* and *C* each have six elements, and four of the corresponding elements are the same while two of them differ. *ipenalty* is 2/6 = 0.33. Using Eq. (8.1), the overall similarity value is scaled by a factor of 1.33 leading to a higher similarity value which denotes a lower degree of similarity.

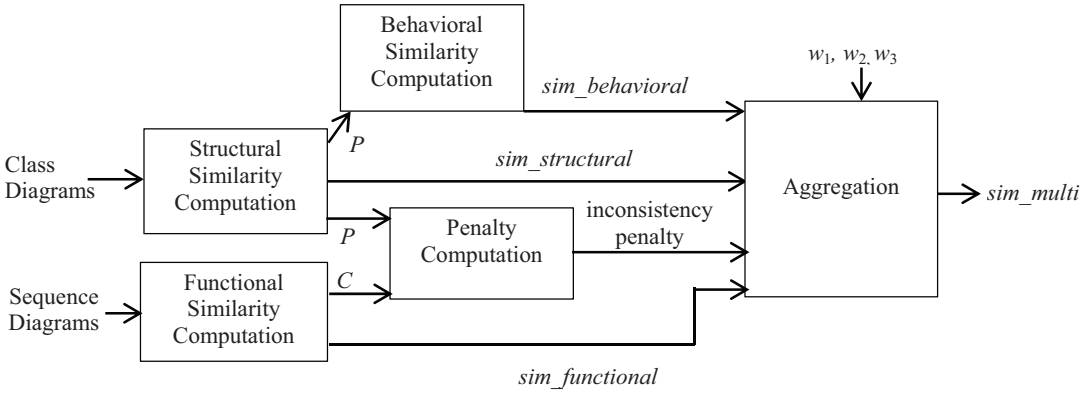


Figure 8.1: Schematic diagram of view composition approach

It is straightforward to show that Eq. (8.1) satisfies three of the four metric axioms: symmetry; minimality; and self-similarity. It has previously been shown that Eq. (5.1), Eq. (6.3) and Eq. (7.2) satisfy these three axioms. Moreover, the last multiplicand in Eq. (8.1) (i.e., 1 + *ipenalty*) satisfies the three metric axioms. It satisfies symmetry, because

the inconsistency penalty does not depend on the order in which two models are presented. The multiplicand satisfies minimality and self-similarity, because for two identical models, *sim_structural*, *sim_functional* and *sim_behavioral* are all zero and there is no inconsistency (i.e., *ipenalty* is zero).

8.2 View Cascading Approach

Multi-view similarity values can also be computed in stages. In each stage, heuristic search techniques are used to obtain suitable mappings of entities belonging to a particular view. Entity mappings in one stage are utilized for computing similarity scores during that stage as well as in all subsequent stages, thus eliminating the need for any inconsistency penalty [5]. The formula for computing multi-view similarity score using cascading is given in Eq. (8.2).

$$sim\_multi = w_1 * sim\_structural + w_2 * sim\_functional + w_3 * sim\_behavioral \quad (8.2)$$

sim_structural, *sim_functional*, *sim_behavioral*, w_1, w_2 and w_3 have the same meanings as the corresponding symbols of Eq. (8.1). As shown in Figure 8.2, the classes mapping (i.e., *P*) for the individual with the best structural similarity score is used as *C* during functional similarity assessment. Thus, in the functional similarity assessment stage, the chromosome represents only the mapping of sequence diagrams (i.e, *S*). Because the size of the search space is reduced in the functional similarity assessment stage, it is expected that this approach will be faster than other multi-view approaches.

Another possible arrangement for view cascading is to use *C* from the functional similarity assessment stage as *P* during structural similarity assessment. One challenge

arising from using this architecture is that some classifiers (e.g. interfaces and abstract classes) in class diagrams may not appear in sequence diagrams. Thus, it is possible that not all classifiers present in the class diagram may be captured in *C*. As a result of this observation, we shall not implement the second possible arrangement for view cascading.

Eq. (8.2) can be considered as a special case of Eq. (8.1) where the inconsistency penalty *ipenalty* equals zero. Thus, the proof that Eq. (8.1) satisfies three of the four metric axioms suffices as a proof that Eq. (8.2) also satisfies the same three axioms.

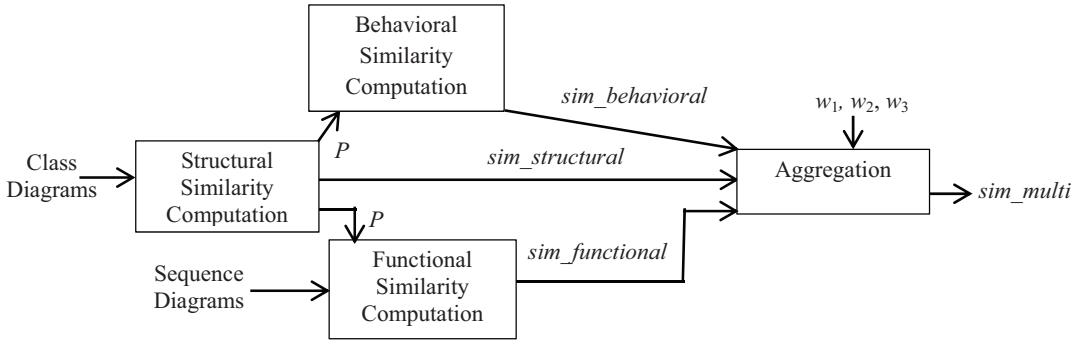


Figure 8.2: Schematic diagram for view cascading

8.3 Simultaneous Search Approach

In the simultaneous search approach, entity mappings in all views are captured in one (large) solution encoding, thus avoiding any inconsistencies across views. In other words, each candidate solution can be seen as a concatenation of *P* and *S*. This is similar to the chromosome encoding used for functional similarity assessment (see Figure 6.6), except that *P* may be larger (i.e., contain more entries) than *C* if there are interfaces or abstract classes in the class diagrams that do not appear in the sequence diagrams. During the search for an optimum solution, Eq. (8.2) is used to evaluate the fitness of each

individual. It is noteworthy that entries in the classifiers' similarity matrix used for population initialization and crossover is computed as a weighted sum of the classifiers' similarity matrix for class diagrams (see Section 5.1.3) and the classes' similarity matrix for sequence diagrams (see Section 6.3.2). Figure 8.3 shows the architecture of the simultaneous search approach.

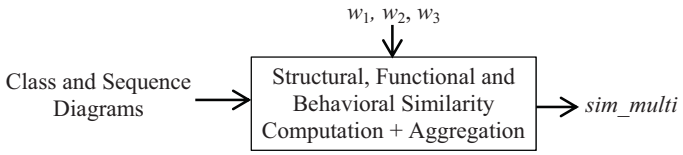


Figure 8.3: Schematic diagram of simultaneous search approach

Chapter 9

Pre-filtering of Repository Models

One of the anticipated gains of software reuse is reduced development time [2]. Reusable software artifacts are usually kept in a repository, from where they can be retrieved for reuse. As the repository increases in size, there is a corresponding rise in retrieval time which can lessen – or in the extreme case outweigh – the expected savings in development time.

This chapter describes a fast way of identifying a subset of repository models which are potentially similar to a query model. Only the shortlisted repository models are compared with the query model in a subsequent computationally demanding retrieval stage to ascertain their actual degree of similarity with the query.

There are two ways of gaining speed-up by selecting a first set of candidate repository models prior to the retrieval stage during what we shall henceforth refer to as the *pre-filtering stage.* First, it leads to significant reduction in retrieval time when there are many projects in the repository and the retrieval stage is time consuming. Secondly, if the relevant metadata about repository projects are extracted or computed beforehand, additional speedup is achieved because it eliminates the need to completely load each repository artifact into primary memory during pre-filtering.

Our method of selecting a subset of repository models involves comparing the metadata of the query to that of all repository projects. The metadata of repository projects are obtained when the artifacts are saved to the repository, while those of the query are gotten as soon as the query is presented to the reuse system. The metadata are *metric-based*, comprising a number of metrics that capture information regarding the size and complexity of UML models.

Software metrics provide quantitative measures of properties (e.g., size, complexity, coupling and cohesion) of software or its specification. A metrics-based similarity score (*MetricSim*) is obtained by comparing metric values of query and repository models. It is expected that corresponding metrics for similar software should not differ significantly. The set of metrics for query and repository models are represented by *k*-dimensional feature vectors, where *k* is the number of metrics. Each dimension of the vector holds the value of a particular metric. *MetricSim* for two software systems is the Euclidean distance between their feature vectors. In order to ensure that features contribute equally to the similarity score, values of each feature are divided by the maximum value of the feature, causing all vectors to contain values between zero and one.

Table 9.1 describes the 16 metrics that form the features of each software system. Typically, there are several sequence diagrams and state machine diagrams for each software. However, because the number of dimensions for the feature vector (i.e., *k*) is fixed, metrics that are applicable to each sequence diagram or state machine diagram are averaged to obtain single values. Similarly, metrics that measure values for classes need to be averaged over the number of classes in the class diagram.

Table 9.1: Description of metrics used for pre-filtering

Metric No.	Metric Description	Reference	Metric is Applicable to:
1	Total number of classifiers in a class diagram.	NC [64]	class diagram
2	Total number of methods in a class diagram	NM [64]	class diagram
3	Total number of attributes in a class diagram		class diagram
4	Total number of associations in a class diagram	NAssoc [64]	class diagram
5	Total number of aggregation relationships diagram (each whole-part pair in an aggregation relationship) in a class diagram	NAgg [64]	class diagram
6	Total number of dependency relationships in a class diagram	NDep [64]	class diagram
7	Total number of generalization relationships (each parent-child pair in a generalization relationship) within a class diagram	NGen [64]	class diagram
8	Total number of generalization hierarchies in a class diagram	NgenH [64]	class diagram
9	Maximum of the Depth of Inheritance Tree (DIT) values obtained for classes in the class diagram. The DIT value for a class within a generalization hierarchy is the longest path from the class to the root of the hierarchy.	MAXIMUM DIT [64]	class diagram
10	Maximum of the HAgg values obtained for classes in the class diagram. The HAgg value for a class within an aggregation hierarchy is the longest path from the class to the leaves.	MAXIMUM HAGG [64]	class diagram
11	Number of messages in a sequence diagram		Sequence diagram[b]
12	Number of classes per use case[a]	[65]	Sequence diagram[b]
13	Number of use cases[a] per class	[65]	Sequence diagram[b]
14	McCabe's cyclomatic complexity	CC [66]	State machine diagram[b]
15	Total number of simple states, considering also the simple states within the composite states.	NS [67]	State machine diagram[b]
16	Number of transitions in a state machine diagram	NT [67]	State machine diagram[b]

a: use case refers to sequence diagrams in our case
b: values will need to be averaged over number of classes, number of state machine diagrams or number of sequence diagrams, as the case may be, in order to obtain a single value for the metric

Chapter 10

Experiments

Empirical results obtained from experiments provide a good way to assess retrieval systems. This chapter discusses the experiments carried out to evaluate the different similarity assessment techniques presented in this dissertation.

Section 10.1 briefly describes our software retrieval tool. Different criteria for evaluating our similarity assessment techniques are described in Section 10.2. The first three experiments, which are described in Sections 10.3, 10.4 and 10.5 evaluate the similarity assessment techniques for individual views i.e., structural, functional and behavioral views. In Sections 10.6, 10.7 and 10.8, we present and discuss the results of experiments that evaluate the multi-view similarity assessment techniques. Section 10.9 focuses on an experiment to assess our pre-filtering technique. Finally, in Section 10.10, some conclusions are drawn from all the experiments.

10.1 Retrieval Tool

The similarity assessment techniques presented in this dissertation were implemented in a retrieval tool, which was developed using the Matlab® computing language. Figure 10.1 shows the query presentation screen. In this screen, the reuser specifies the XMI file for the query, and the folder/directory containing XMI files for the requirement

specifications of repository projects. The reuser can also specify whether pre-filtering should be carried out before retrieval or not. If pre-filtering is required, the number of repository projects to be shortlisted is specified.

Figure 10.2 shows the list of repository projects, ranked according to their degree of similarity with the query. The reuser can choose the top ranking project, or any other project from the list. The artifacts for the selected project are duplicated in the repository. The reuser modifies the artifacts of the replicated project to meet the needs of the system being developed.

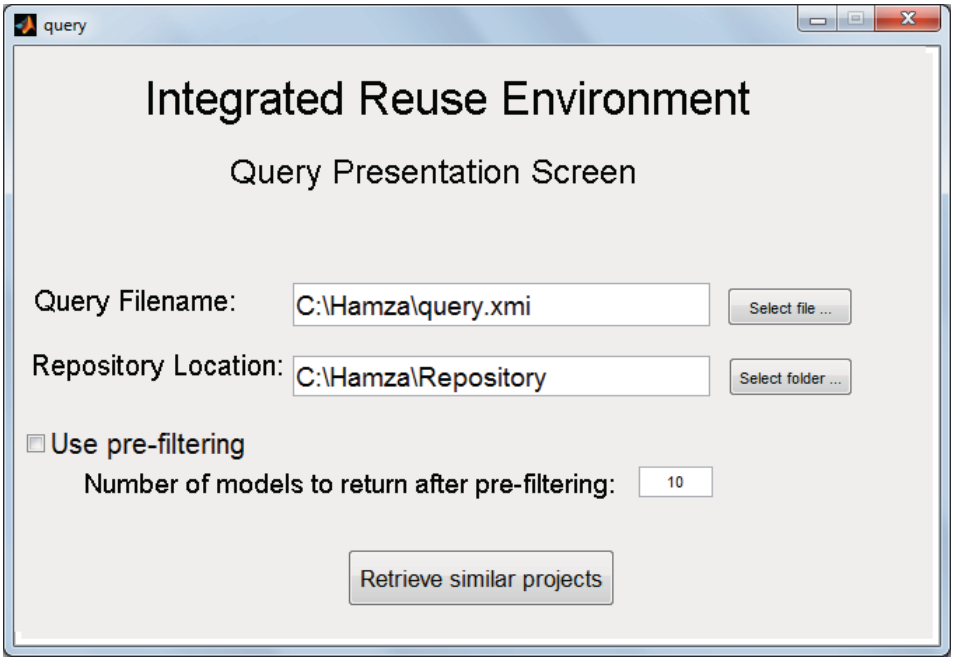


Figure 10.1: Query presentation screen

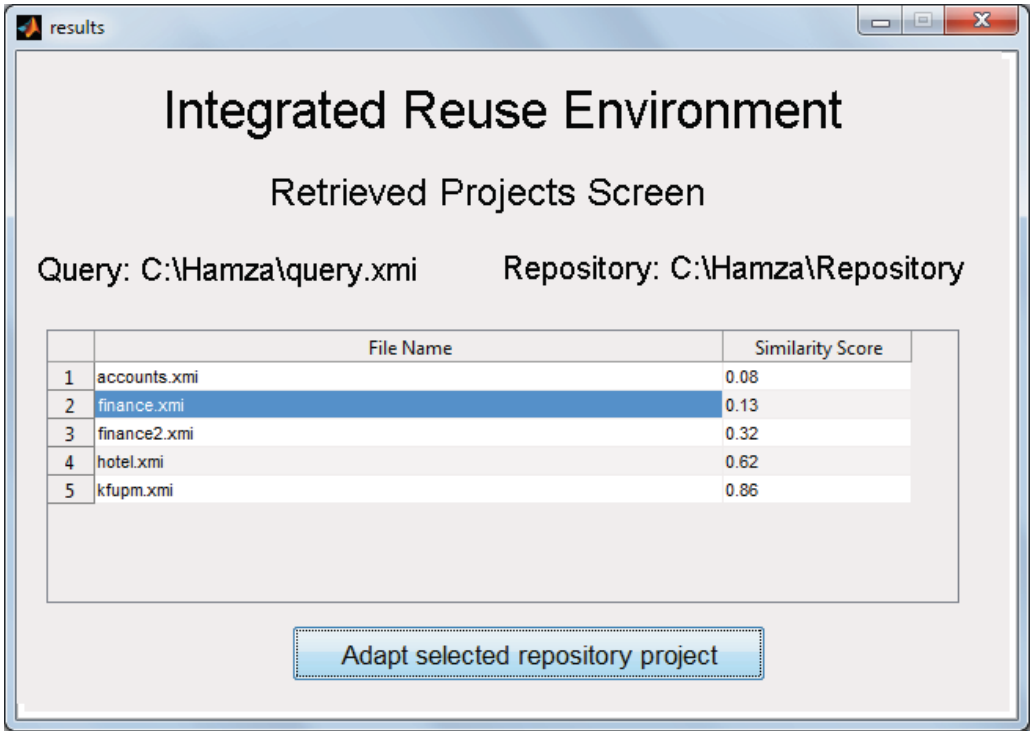


Figure 10.2: List of projects returned after similarity assessment

10.2 Evaluation Criteria

In this section, we describe measures for assessing our reuse technique. All experiments were carried out on a personal computer having the following configuration: 2.67 GHz Intel Core 2 Quad processor; 4 GB RAM; and 32-bit Windows 7 operating system.

10.2.1 Retrieval Quality

Several measures that are commonly used to assess the retrieval quality in the IR field are described in this section.

Recall

Recall is the proportion of relevant repository artifacts that are retrieved. The formula for recall is given in Eq. (10.1). In order to obtain high recall, a retrieval system should return

as many relevant artifacts as possible. For example, a user who wants to find all patents or law suits in a particular area will be interested in high recall.

$$\text{Recall} = \frac{\text{Number of relevant and retrieved artifacts}}{\text{Number of relevant artifacts in the repository}} \tag{10.1}$$

Precision

Precision is the proportion of retrieved artifacts that are relevant to the query. The formula for precision is given in Eq. (10.2). High precision can be obtained by minimizing the number or irrelevant artifacts that are returned. For example, a user who looks for information using a web search engine is usually interested in obtaining as many relevant results in the first page as possible (i.e., high precision). S/he may not have an idea of all relevant documents in the area, and may not even be interested in retrieving all of them [17].

$$\text{Precision} = \frac{\text{Number of relevant and retrieved artifacts}}{\text{Number of retrieved artifacts}} \tag{10.2}$$

F Measure

Recall and precision tradeoff against each other. For example, high recall but low precision can be obtained by retrieving all repository models. On the other hand, high precision but low recall can be obtained by not retrieving any repository models [17, 68]. The F-measure is the weighted harmonic mean of precision and recall. It combines both measures into one value, depending on their relative importance. The formula for F-measure is given in Eq. (10.3):

$$F_\alpha = \frac{\text{Precision} \cdot \text{Recall}}{(1-\alpha)\text{Precision} + \alpha \cdot \text{Recall}} \tag{10.3}$$

where $\alpha \in [0, 1]$ is a parameter that determines the relative importance of recall with respect to precision. The F-measure is commonly used with $\alpha = 0.5$ [69].

All the three IR measures described above are suitable for unranked or set-based retrieval because the measures are computed mainly from the number of relevant documents that are retrieved. In the sequel, we discuss other measures which have been specifically adapted for ranked retrieval. These later measures take into account, the order in which returned documents are presented.

Mean Average Precision (MAP)

The average precision (AP) for a query is obtained using precision values calculated at each point when a new relevant document is retrieved (using precision = 0 for each relevant document that was not retrieved). Mean Average Precision, also referred to as *mean precision at seen relevant documents* for a set of queries is the mean of the AP scores for each query [69]. The formula for MAP is given in Eq. (10.4):

$$MAP = \frac{1}{N}\sum_{j=1}^{N}\frac{1}{Q_j}\sum_{i=1}^{Q_j}P(rel = i) \tag{10.4}$$

where N is the number of queries, Qj is the number of relevant documents for query j and $P(rel = i)$ is the precision at the i^{th} relevant document.

Precision at K

Another ranked retrieval measure is *Precision at K*. This is the precision after the top K documents are considered. Unfortunately, because K is a constant, this measure is highly unstable and depends on the number of relevant documents. Assume R is the number of

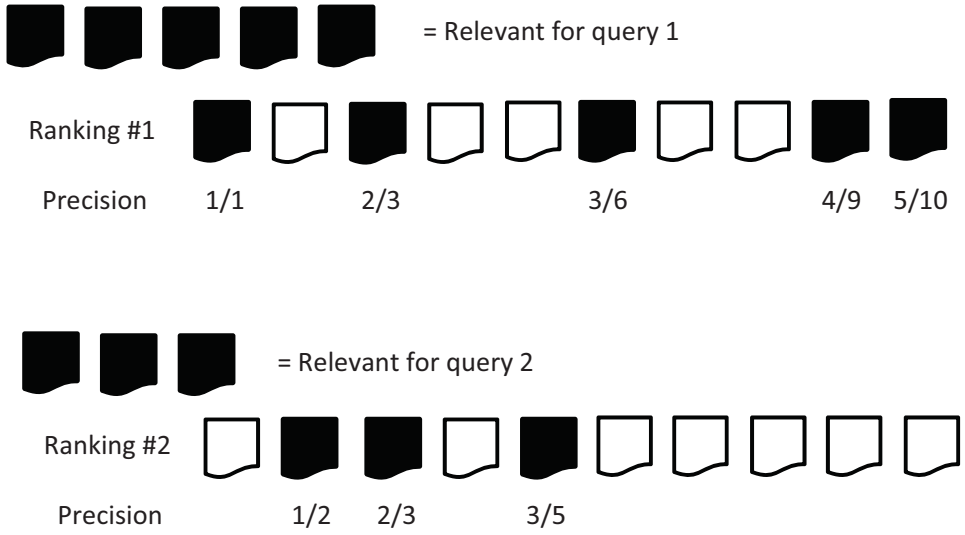
relevant documents. If $R > K$, a perfect system will always return a recall value that is less than 1. On the other hand, if $R < K$, a perfect system always returns a precision value that is below 1.

R Precision

R Precision is the same as *Precision at K*, when *K* is the number of documents that are relevant to a query. *R Precision* is an interesting measure because precision is equal to recall at the R^{th} position.

We shall measure the efficiency of our retrieval methods using the MAP because it is widely used to evaluate ranked IR systems. Furthermore, unlike *R Precision* and *Precision at K*, MAP produces a single value when there are multiple queries.

Figure 10.3 illustrates how MAP is computed. There are ten documents in the repository and two queries. Five documents are relevant to the first query whereas three documents are relevant to the second query. The mean of AP for both queries is returned as the MAP.



Average Precision Query 1 = (1/1 + 2/3 + 3/6 + 4/9 + 5/10)/5 = 0.62

Average Precision Query 2 = (1/2 + 2/3 + 3/5)/3 = 0.59

Mean Average Precision for both queries = (0.62 +0.59)/2 = 0.60

Figure 10.3: Computation of MAP

10.2.2 Correlation between Similarity Scores and Estimated Reuse Effort

Even though a reuse system is able to retrieve relevant projects from a repository with high MAP, it is possible that it is only good at ranking the repository projects but the similarity scores themselves are meaningless. To address this possibility, we shall examine the degree of correlation between the similarity scores returned by our reuse system and estimated modification (reuse) effort. Since a significant amount of reuse effort is dedicated to programming, code-based sizing metrics will be used to estimate reuse effort. We shall use the same formula employed by Basili et al. [70] for predicting software maintenance effort. Effort (in man hours) is estimated as follows [70]:

Effort = 0.36 * effective SLOC + 1040.

Where *effective source lines of code* (*SLOC*) is the sum of added, deleted and modified SLOC. Effective SLOC is computed using Unified Code Counting tool[1]. A strong degree of correlation between similarity scores and estimated reuse effort shows that similarity scores returned by the reuse system can provide a reuser with a rough estimate of the amount of effort needed to adapt retrieved software artifacts to suit the needs of the software system being developed. It is important to note that correlation between similarity scores and reuse effort is computed only for experiments that use a dataset for which the source code is available.

10.2.3 Retrieval Time

One of the expected gains of reuse is reduction in development time. A retrieval method which has excellent precision and recall based values, but which needs unacceptably long time to execute will not be utilized by any reuser. Thus, we shall also measure the retrieval time of our system.

10.3 Experiment 1

This section presents and discusses an experiment carried out to evaluate the structural similarity assessment techniques described in Chapter 5. The two objectives of the experiment were: (i) to compare deep and shallow structural similarity assessment techniques (ii) to compare the performance of GA and CSA in shallow structural similarity assessment.

[1] http://sunset.usc.edu/research/CODECOUNT/

10.3.1 Experimental Data

Data scarcity is a common problem for software engineering research [71]. Because there are no available software reuse repositories containing UML diagrams, we relied on reverse engineered class and sequence diagrams for six families of open source software. Altova® UModel®[2] was used to reverse engineer source code to UML diagrams. The repository contained five versions of each software family, making a total of 30 projects. Furthermore, 30 queries Q_1 … Q_{30} were formed by using each of the repository models in turn (i.e, $Q_i = R_i$, $1 \leq i \leq 30$). The similarity between each query model and every repository model was determined. Intuitively, R_i is relevant to Q_i ($1 \leq i \leq 30$) only if Q_i and R_i are versions of the same software family. For example, $R_1 \ldots R_5$ are relevant to $Q_1 \ldots Q_5$, while $R_{26} \ldots R_{30}$ are relevant to $Q_{26} \ldots Q_{30}$. A brief description of the various software families is presented in Table 10.1.

[2] http://www.altova.com/umodel.html

Table 10.1: Description of software systems used for experiments

Software Family	Java Game Maker[3] (JGM)	Plot Digitizer[4] (PD)	Open Stego[5] (OG)	JOrtho[6] (JO)	51 Degrees[7] (51D)	Jcurses[8] (JC)
Brief Description	game engine for developing java games	For digitizing data points off of scanned plots, scaled drawings, etc.	steganography tool	Java based spell checker	For detecting mobile devices that browse a website	Console toolkit for Windows®
Versions	1.9, 2.1, 2.2, 2.9, 3.1	2.3.0, 2.4.1, 2.5.0, 2.6.0, 2.6.2	0.2.0, 0.3.0, 0.4.0, 0.5.0, 0.5.2	0.2, 0.3, 0.4, 0.5, 1.0	2.2.8.5, 2.2.8.6, 2.2.8.7, 2.2.8.8, 2.2.8.9	0.91, 0.92, 0.94, 0.95, 0.95b
Label in repository	$R_1 – R_5$	$R_6 – R_{10}$	$R_{11} – R_{15}$	$R_{16} – R_{20}$	$R_{21} – R_{25}$	$R_{26} – R_{30}$
No. of classifiers in class diagrams	27 – 37	44 – 66	11 – 59	22 – 56	52	57 - 66
No. of sequence diagrams	54 – 98	48 – 69	15 – 92	66 – 88	77 – 84	178 - 254
No. of messages in all sequence diagrams	581-1838	648 – 942	172 – 3895	419 – 2175	1331 - 1366	4991 – 9291

[3] http://sourceforge.net/projects/java-game-maker/?source=directory
[4] http://sourceforge.net/projects/plotdigitizer/?source=directory
[5] http://sourceforge.net/projects/openstego/?source=directory
[6] http://sourceforge.net/projects/jortho/
[7] http://sourceforge.net/projects/fiftyone-java/
[8] http://sourceforge.net/projects/javacurses/?source=directory

10.3.2 Results and Discussion

Table 10.2 lists the parameters used for Experiment 1. The values of α, w_1, w_2, w_3, w_4 and w_5 were obtained by systematically trying different values.

Table 10.2: Parameters for Experiment 1

	Parameter Description	Value
Shallow similarity assessment	Size of population/ number of nests	50
	Maximum number of generations	1000
	Number of generations to terminate algorithm if fitness value does not improve	20
	Number of individuals from initial population produced using Munkres' algorithm	3
	Probability of mutation (GA)	0.025
	Fraction of nests to abandon (CSA)	0.3
	Number of repetitions	30
	α in Eq. (5.1)	0.05
Deep similarity assessment	w_1 in Eq. (5.3)	0.7
	w_2 in Eq. (5.5)	0.3
	w_3 in Eq. (5.5)	0.1
	w_4 in Eq. (5.7)	0.4
	w_5 in Eq. (5.7)	0.2

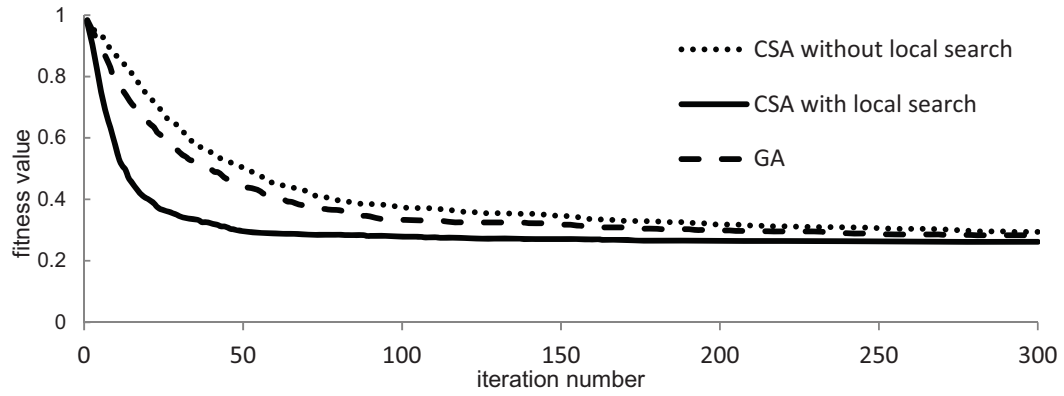
The convergence characteristics of GA and CSA are shown in Figure 10.4 for two pairs of query and repository diagrams (Q_{11}, R_{15}) and (Q_6, R_9). It can be observed that in both cases, CSA converges faster when local search is performed during Lévy flight. During each step of the flight, the fitness of a nest is evaluated to determine if it is better than all the previous nests formed in the flight. If the current nest is better than previous nests, it is recorded as the best nest found so far in the flight. In Figure 10.4 (a), CSA with local

search converges faster than GA. However, in Figure 10.4 (b), GA converges faster than both forms of CSA. Figure 10.5 shows the proportion of time when GA and CSA (with local search) return a lower fitness value than each other for all comparisons of query and repository diagrams. It can be seen that half the time, CSA produces a better fitness value than GA. However, this comes with a price; as shown in Table 10.3, GA runs approximately 3.2 times faster than CSA. GA and CSA require 24.06 and 76.90 seconds, respectively, to search the repository. As expected, deep similarity assessment required the largest amount of time (120.06 seconds), since it involved more detailed computations than shallow similarity assessment.

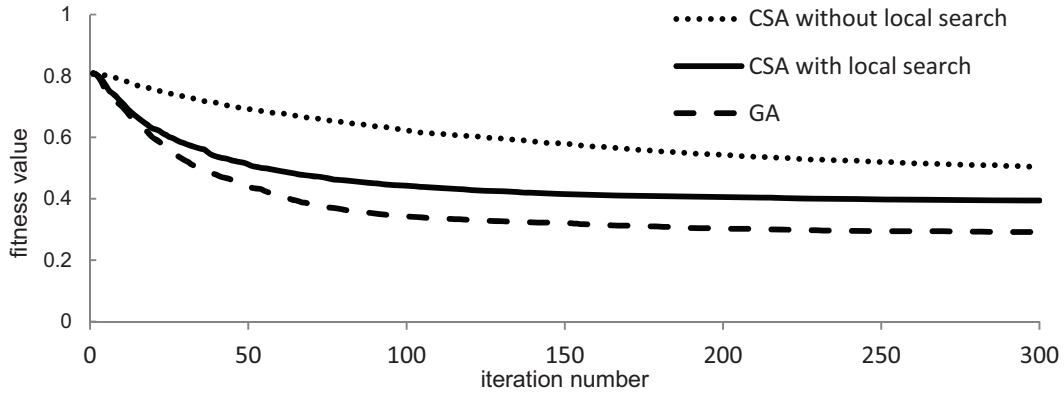
Values of MAP for all queries are shown in Table 10.3. The mean of MAP over thirty runs are reported for both GA and CSA, while the standard deviation is shown in parentheses. From the results, it can be seen that deep and shallow similarity assessment of class diagrams give very good results, since the most relevant class diagrams are returned as top ranking diagrams when a query is presented. The values of MAP are at least 97% for all methods.

Both GA and CSA have correlation coefficients of about 82% which is significant at 10^{-101} level. Since the degree of correlation is strong, it indicates that similarity scores returned by both algorithms can provide a reuser with a rough estimate of the amount of effort needed to adapt retrieved software artifacts to suit the needs of the software system being developed. Deep similarity scores have a 75.13% correlation with estimated reuse effort which is significant at 10^{-85} level.

From the presented results, deep and shallow similarity assessment produces more or less the same MAP values. However, compared to deep similarity assessment, shallow similarity assessment results in better correlation with predicted reuse effort.



(a)



(b)

Figure 10.4: Convergence characteristics of GA and CSA for (a) Q_{11}, R_{15} (b) Q_6, R_9

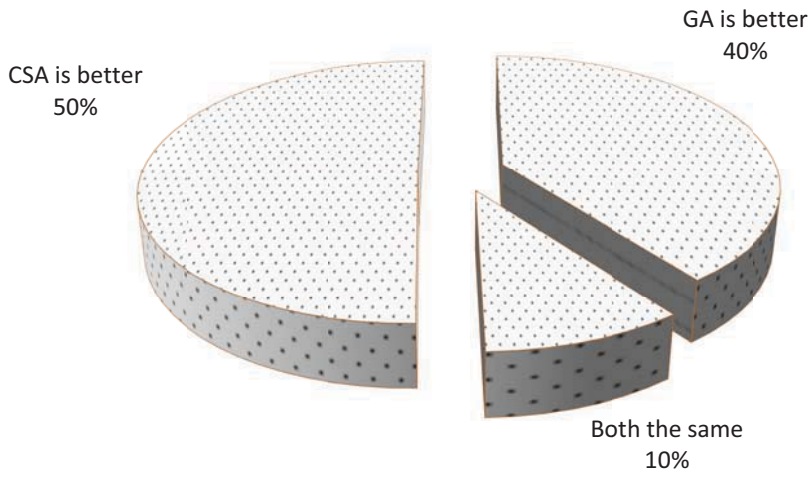


Figure 10.5: Proportion of time when GA and CSA produce better fitness values than each other

Table 10.3: Comparison of results for CSA and GA

	MAP (%)	Correlation with reuse effort (%)	time to search repository (seconds)
Shallow Similarity (GA)	97.81 (0.83)	82.77 (0.63)	24.06
Shallow Similarity (CSA)	97.73 (0.78)	83.92 (0.42)	76.90
Deep Similarity	97.81	75.13	120.06

10.3.3 Conclusion

Experimental results show that the structural similarity assessment techniques introduced in this dissertation produce excellent results. Apart from the high values of MAP obtained, there is also a strong correlation between similarity scores and estimated reuse effort.

Deep similarity assessment of class diagrams produced very high values of MAP like shallow similarity assessment. However, deep similarity assessment required more time and produced lower correlation with reuse effort, compared to shallow similarity assessment.

Even though shallow similarity assessment using GA and CSA produced more or less the same results, the former incurred less retrieval time than the latter, thus it is recommended to use GA-based shallow similarity assessment when retrieving the most similar projects from a repository based on the similarity of class diagrams.

10.4 Experiment 2

This section describes an experiment carried out to evaluate our proposed methods of determining the similarity of two sequence diagrams. The objectives of the experiment were twofold: to compare the graph-based and LCSMM-based functional similarity assessment; and to compare the performance of GA and CSA.

10.4.1 Experimental Data

We chose 10 query sequence diagrams (q_1 … q_{10}) from two sources: eight queries were obtained from software engineering undergraduate lecture materials; while two queries were taken from textbook examples found in Yue et al. [72]. As shown in Table 10.4, the sequence diagrams have between 6 and 27 messages, and involve between 3 and 9 objects.

Different sets of repositories containing 60 sequence diagrams were created such that each query partially matched six repository diagrams. Let m_i denote 30% of the number of messages in q_i. The six repository diagrams that match q_i are constructed in one of three ways: randomly deleting m_i messages from q_i; randomly adding messages so that the repository diagram has m_i more messages than q_i; and randomly adding or deleting messages to/from the query m_i different times to form a repository diagram. The choice of 30% of query diagram messages was to ensure that query and relevant repository

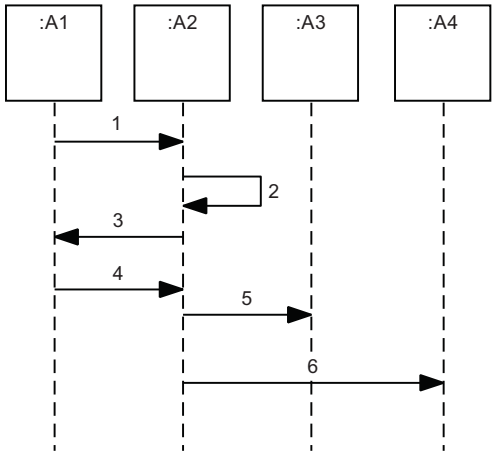
diagrams differ in small but significant ways. The experiment was carried out using ten sets of repositories generated in the manner described above. Figure 10.6 shows a query sequence diagram (q_3) and three repository diagrams that were randomly created from q_3. The rounded rectangles indicate where the repository diagrams differ from the query diagram.

Table 10.4: Properties of query sequence diagrams

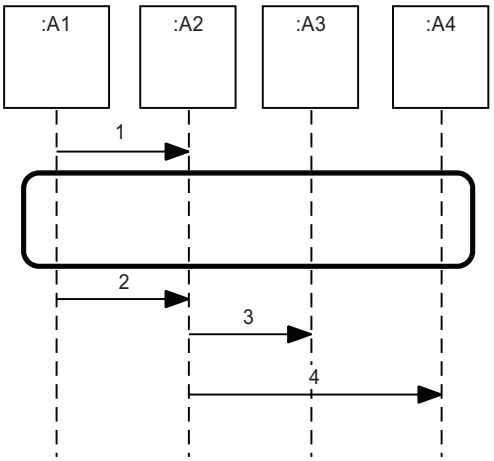
Query	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}
Number of messages	9	6	6	11	10	17	11	8	15	27
Number of objects	6	6	4	5	5	3	6	4	7	9

10.4.2 Results

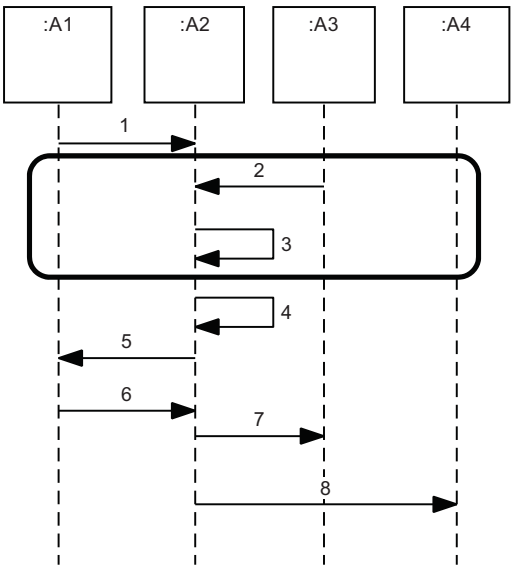
Values of parameters used to test our methods are shown in Table 10.5. We compared our work with the sequence diagram similarity scoring approach of Park and Bae [37]. They determined the similarity between two sequence diagrams using an iterative graph similarity algorithm. The algorithm computes a similarity score between two Message-Object-Order-Graphs (MOOGs) based on the idea that graph elements (nodes or edges) in two graphs are similar if their respective neighborhoods are similar. First, both sequence diagrams are converted to MOOGs in which there are nodes whenever messages are sent or received, and there are edges denoting message flow between objects and time flow within objects. The values of parameters and number of iterations were the same as those used in [37].



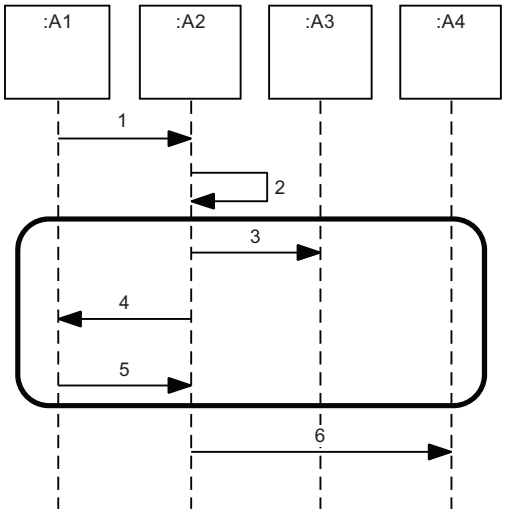
(a) query diagram



(b) repository diagram formed by deleting two messages from query



(c) repository diagram formed by adding two messages to query



(d) repository diagram formed by adding and/or deleting messages from query two times

Figure 10.6: Creation of repository diagrams from a query

Table 10.5: Parameters for Experiment 2

Parameter Description	GA-MOOG	CSA-MOOG	GA-LCSMM	CSA-LCSMM
Size of population/ number of nests	50	50	10	10
Maximum number of generations	1000	1000	500	500
Number of generations to terminate algorithm if fitness value does not improve	20	20	20	20
Number of individuals from initial population produced using Munkres' algorithm	3	3	0	0
Probability of mutation	0.02	-	0.1	-
β in Eq. (6.1)	0.15	0.15	-	-
Fraction of nests to abandon	-	0.3	-	0.3
Number of repetitions	20	20	20	20

Figure 10.7 shows the mean and standard deviation of MAP for the five methods (Park and Bae, GA-MOOG, CSA-MOOG, GA-LCSMM and CSA-LCSMM). The experiment was repeated 20 times for all methods except Park's, which is a deterministic algorithm. The low values of MAP produced by Park's method indicate that it did not usually return relevant repository diagrams as the top ranking diagrams in response to the queries. It can be seen that our method of similarity assessment using MOOGs returned most of the relevant repository diagrams in response to the queries. Furthermore, the LCSMM-based retrieval technique found all the relevant repository diagrams for each query except for a few cases.

Another advantage of the LCSMM-based method is that it runs much faster than the MOOG-based method. Both methods have lower execution time compared to Park's method. Figure 10.8 shows the retrieval time for each of the methods. GA-LCSMM and CSA-LCSMM require 0.90 and 1.55 seconds to search the repository, respectively.

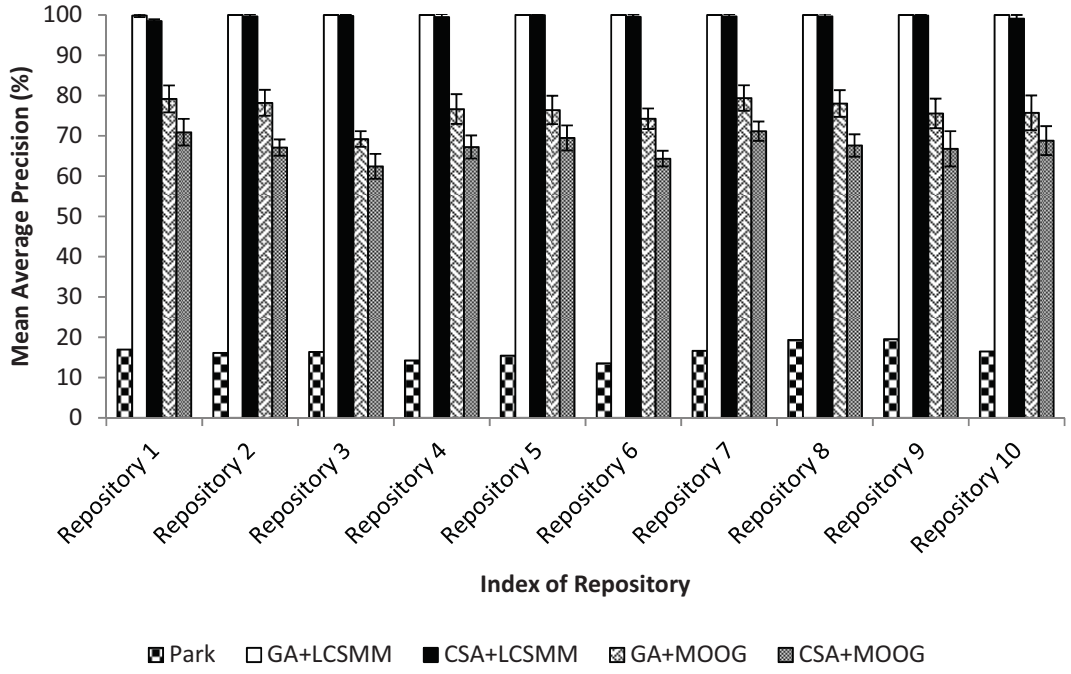


Figure 10.7: Mean and standard deviation of MAP for the five methods

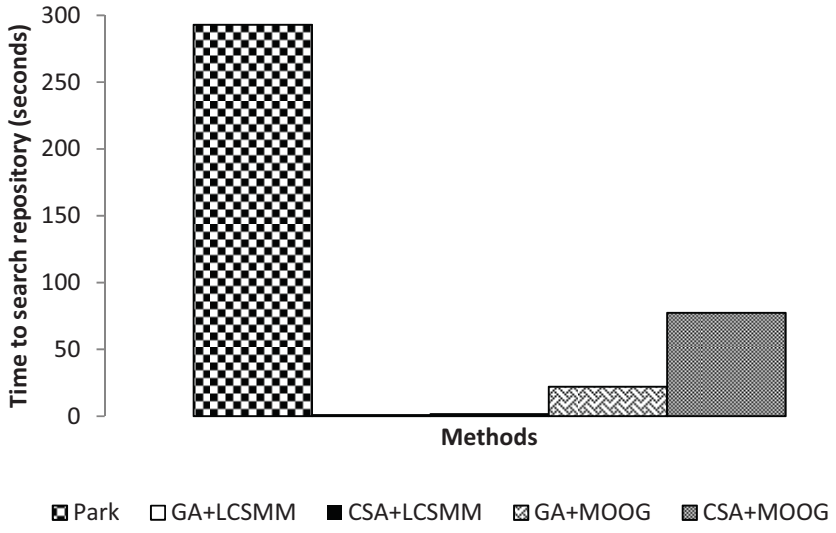


Figure 10.8: Time to search the repository for the five methods

In order to investigate the poor performance of Park's method, we computed the pairwise similarity values for the three sequence diagrams of Figure 6.1. The similarity values

obtained by using GA-MOOG, GA-LCSMM and Park's method are shown in Figure 10.9.

	a	*b*	*c*
a	5.59	6.48	6.46
b	6.48	7.72	7.89
c	6.45	7.89	8.48

(a) Park

	a	*b*	*c*
a	0.00	0.15	0.38
b	0.15	0.00	0.50
c	0.39	0.50	0.00

(b) GA-MOOG

	a	*b*	*c*
a	0.00	0.33	0.33
b	0.33	0.00	0.50
c	0.33	0.50	0.00

(c) GA-LCSMM

Figure 10.9: Pairwise similarity of sequence diagrams of Figure 6.1 using (a) Park's method (b) GA with MOOG (c) GA with LCSMM

10.4.3 Discussion of Results

The results presented in Figure 10.7 indicate that our methods were able to retrieve the most relevant sequence diagrams in response to a given query diagram. The LCSMM-based method returned all relevant repository diagrams as the top ranking diagrams in almost all cases. The method based on GA and MOOGs returned most, but not all the relevant repository diagrams. On the other hand, Park's method was unable to retrieve the most relevant diagrams. It is noteworthy that experiments that used GA for matching gave slightly better MAP compared to the corresponding ones that used CSA.

Figure 10.8 shows that LCSMM-based retrieval was faster than MOOG-based retrieval, which was in turn much faster than Park's method of retrieval. The GA-based experiments were faster than those that used CSA for matching.

One reason why LCSMM-based retrieval performs better than MOOG-based retrieval is that the search space for the former is usually significantly smaller than that of the latter. With the LCSMM-based method, the size of an encoded solution (chromosome or nest) is

the same as the smaller of the number of classes in the two sequence diagrams, whereas with the MOOG-based method this size is twice the number of messages contained in the sequence diagram having fewer messages. For example, if the two sequence diagrams to be compared each have ten messages and involve four classes, the length of the solution for the LCSMM-based and MOOG-based methods are four and twenty, respectively. Recall that class mappings are sufficient to find the LCSMM, whereas in MOOGs, there are two nodes for each message denoting when a message is sent and when it is received.

It can be observed from Figure 10.9 that Park's method and ours satisfy the symmetry axiom (see Sections 6.1.2 and 6.2.2). Moreover, we examined Figure 10.9 for the axiom of self-similarity. Our methods (MOOG-based and LCSMM-based) satisfy this axiom since all values in the leading diagonal are the same. On the other hand, Park's method clearly does not satisfy this axiom. Finally, we scanned the similarity scores to determine if all methods satisfy the axiom of mimimality. Surprisingly, we noted that Park's method does not always give intuitive results. It is expected that the similarity scores in the leading diagonal should have the largest values in each row because in their method, higher similarity scores denote higher degrees of similarity. However, in some rows there are entries (which are encircled in Figure 10.9(a)) that have higher values than the diagonal elements. In contrast, our methods satisfy the minimality axiom because the diagonal entries of Figure 10.9(b) and Figure 10.9(c) have the smallest possible value (that is, zero). Thus, our similarity measures for two sequence diagrams satisfy the self-similarity, minimality and symmetry axioms, while Park's method satisfies only the latter axiom.

The poor experimental results obtained using Park's method can be attributed to two factors. Firstly, their method is a two-stage process. In the first stage, a number of repository models are selected based on the structural similarity between query and repository class diagrams. Sequence diagrams from the shortlisted repository models are then compared in the second stage. However, we have compared our method with only the second stage of their method. It can be argued that the first stage of Park's method is a filtering stage which can potentially sieve out irrelevant projects before sequence diagram comparison takes place.

Secondly, Park computes the similarity of MOOGs using Zager's graph matching algorithm [73], which performs normalization at the end of each iteration, leading to similarity scores that may not produce useful similarity information [74]. This also explains why GA-MOOG and CSA-MOOG perform much better than Park's method, even though they all rely on the same graphical representation of sequence diagrams.

10.4.4 Conclusion

It is recommended to use the LCSMM-based sequence diagram similarity assessment technique with GA, since it resulted in the highest MAP and required the least retrieval time among all the methods that were compared.

10.5 Experiment 3

This section describes an experiment to evaluate our method of similarity assessment of two state machine diagrams.

10.5.1 Experimental data

We used a repository of 16 state machine diagrams belonging to three domains: 7 diagrams are from the banking/business domain; 6 diagrams are from the education domain; while the remaining 3 diagrams are related to time management tasks such as appointment and diary management. The diagrams were obtained from undergraduate and graduate students' course projects and thesis. Table 10.6 summarizes the characteristics of the repository diagrams. 16 queries were formed by taking each of the repository diagrams in turn. A repository diagram is relevant to a query if they belong to the same domain.

Table 10.6: Properties of state machine diagrams in the repository

	Banking/Business							**Education**						**Personal Organization**		
No. of states	10	4	4	6	5	8	5	10	7	8	3	6	6	5	5	5
No. of transitions	14	4	4	5	8	15	5	14	12	9	2	7	6	8	8	4

10.5.2 Results and Discussion

The following parameters were used: size of population = 50; maximum number of generations = 100; number of generations to terminate GA if fitness value does not improve = 20; probability of mutation of genes = 0.10; number of individuals from initial generation produced using Munkres' algorithm = 3. In addition, λ was set to 0.05 in Eq. (7.1). The experiment was repeated 30 times. Table 10.7 shows the mean MAP for the 16 queries. The standard deviation of MAP is shown in brackets. The time to search the repository is also presented in the table.

Table 10.7: Results for behavioral similarity assessment

MAP (%)	time to search repository (seconds)
73.27 (0.24)	1.32

It can be observed from Table 10.7 that our method effectively retrieves similar software from the repository by comparing the software's state machine diagrams.

10.5.3 Conclusion

The behavioral similarity assessment technique described in this dissertation leads to the retrieval of relevant projects from a repository.

10.6 Experiment 4

This section describes an experiment to evaluate the different multi-view approaches.

10.6.1 Experimental Data

The dataset of Section 10.3.1 was used. Because the dataset does not contain state machine diagrams, only the structural and functional components of the multi-view similarity scores were computed. The aim of the experiment was to compare single view (i.e., structural view and functional view) similarity assessment and 2-view (i.e., structural and functional views combined) multi-view similarity assessment.

10.6.2 Results

Three different combinations of w_1 and w_2 were used to aggregate the structural and functional similarity values in order to obtain an overall multi-view score using equations (8.1) and (8.2). In all cases, w_3 was set to zero since the dataset did not contain state

machine diagrams. Table 10.8 provides some brief information on the similarity assessment techniques to be compared.

The parameters used to run the multi-view experiments are shown in Table 10.9. The mean of MAP and coefficient of correlation between similarity scores and reuse effort are shown in Figure 10.10 and Figure 10.11, respectively. In addition, the average time required to search a repository of 30 projects for the different methods are presented in Figure 10.12

Table 10.8: Summary of similarity assessment techniques to be compared

Notation	Description	w_1	w_2	Section of dissertation described
STRUC	(Shallow) structural similarity score	-	-	05.1
FXN	Functional similarity score	-	-	6.3
COMP0.25	Multi-view similarity score using composition	0.25	0.75	8.1
COMP0.5	Multi-view similarity score using composition	0.50	0.50	8.1
COMP0.75	Multi-view similarity score using composition	0.75	0.25	8.1
CASC0.25	Multi-view similarity score using cascading	0.25	0.75	8.2
CASC0.5	Multi-view similarity score using cascading	0.50	0.50	8.2
CASC0.75	Multi-view similarity score using cascading	0.75	0.25	8.2
SIMULT0.25	Multi-view similarity score using simultaneous search	0.25	0.75	8.3
SIMULT0.5	Multi-view similarity score using simultaneous search	0.50	0.50	8.3
SIMULT0.75	Multi-view similarity score using simultaneous search	0.75	0.25	8.3

Table 10.9: Parameters for Experiment 4

Parameter Description	**Value**
Size of population	50
Maximum number of generations	1000
Number of generations to terminate algorithm if fitness value does not improve	20
Number of individuals from initial population produced using Munkres' algorithm	3
Probability of mutating class genes	0.01
Probability of mutating sequence diagram genes	0.01
Number of repetitions	5
α in Eq. (5.1)	0.05
γ in Eq. (6.3)	0.15
(w_1, w_2, w_3) in equations (8.1) and (8.2)	(0.25, 0.75, 0), (0.5, 0.5, 0), (0.75, 0.25, 0)

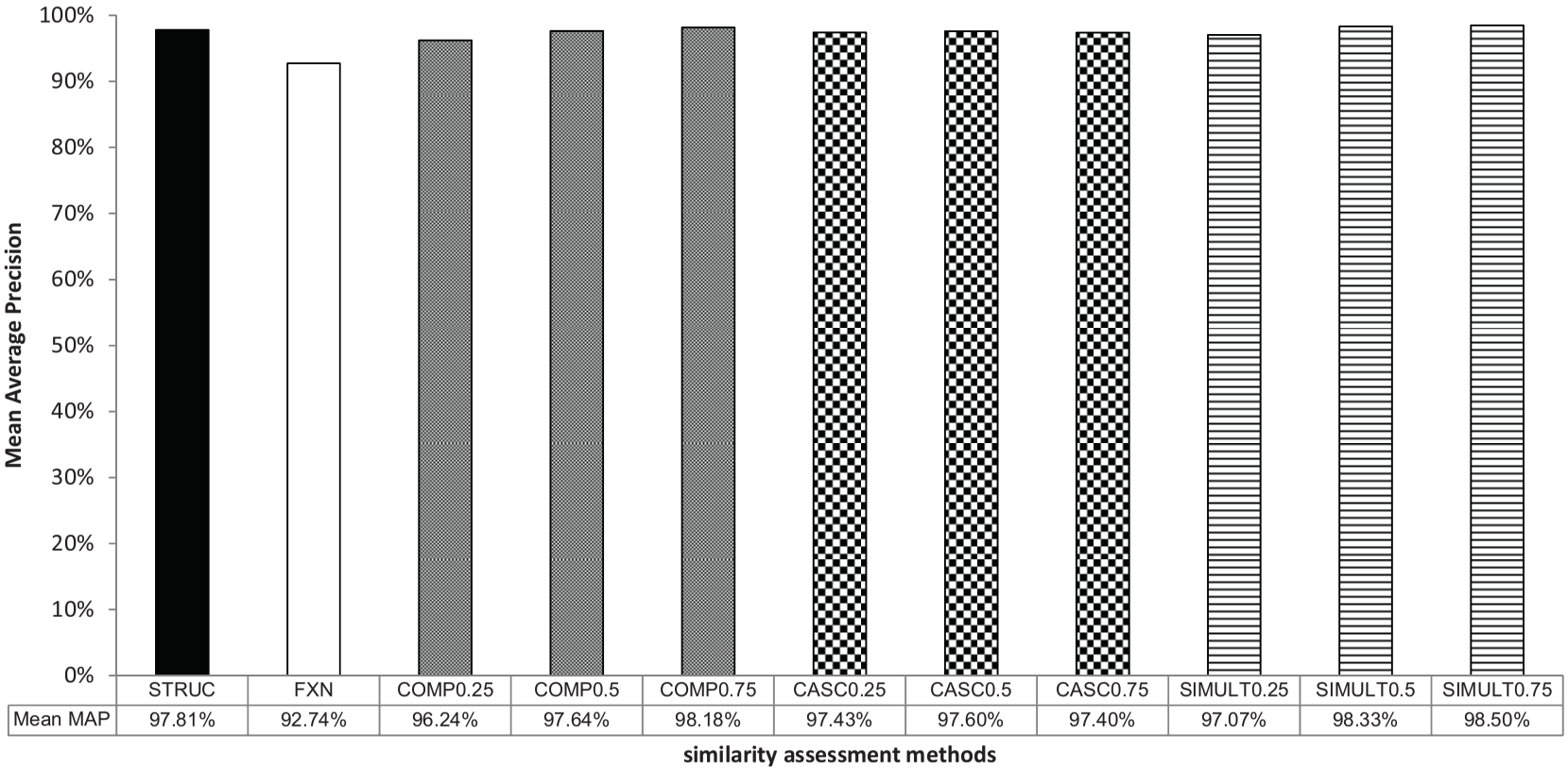


Figure 10.10: MAP for all methods

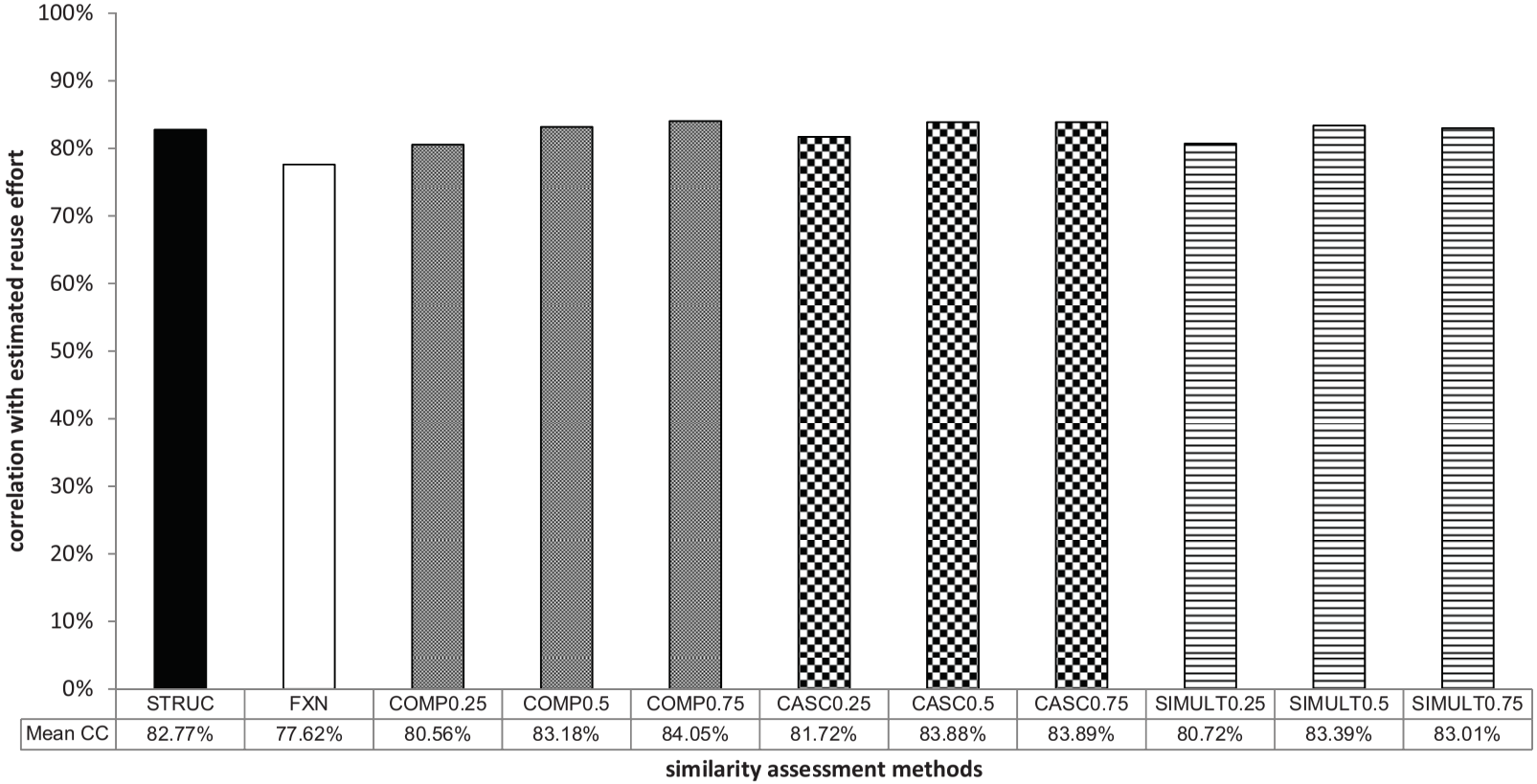


Figure 10.11: Correlation with reuse effort for all methods

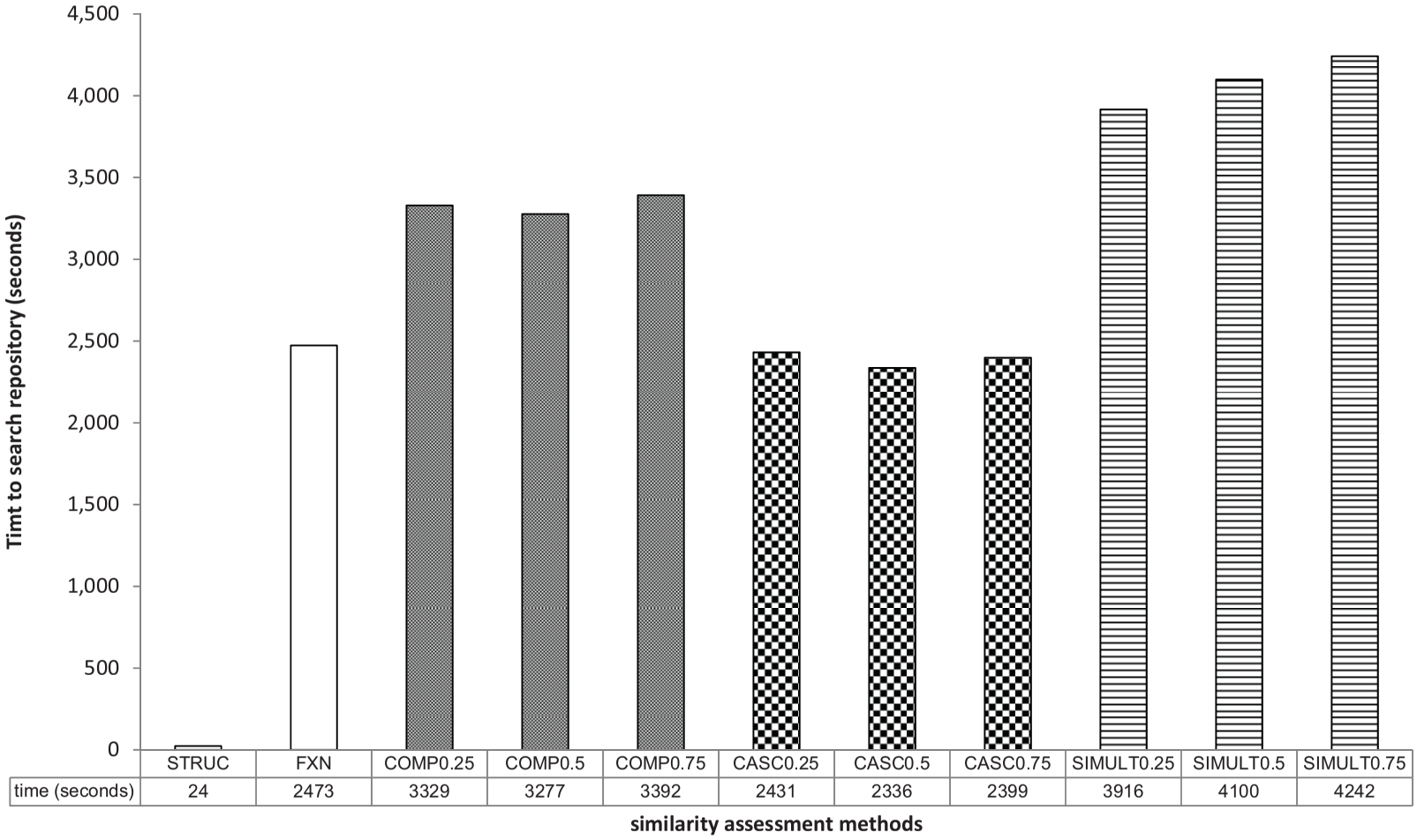


Figure 10.12: Time to search the repository for all methods

10.6.3 Discussion of Results

Values of MAP, retrieval time and correlation with reuse effort for the structural similarity assessment technique are either the best or among the best compared to all the other methods. The excellent results obtained by using *STRUC* are due to the following reasons: (i) because only classifiers are matched during structural similarity assessment, *STRUC* involves the smallest search space among all the methods. As a result, it requires only a small fraction of the retrieval time of the other methods. It is common to have only one class diagram, but many sequence diagrams in a project. (ii) The *Diff* matrix (see Table 5.2) used for computing structural similarity scores contains values that reflect the degree of (dis)similarity of different class diagram relationships. The matrix was adapted from the work of Robles et al. [39], who obtained the dissimilarity information by interviewing UML experts. This contributed to the strong degree of correlation between similarity values and estimated reuse effort. It also aided *STRUC* to retrieve almost all relevant projects from the repository in response to each of the queries, resulting in a MAP of 97.81%.

The results for functional similarity assessment are very good, but do not match those of *STRUC*. *FXN* required much more time for retrieval than *STRUC* because the search space was much larger; sequence diagrams needed to be matched in addition to classes. MAP for *FXN* was 92.74%, but it was less impressive than the corresponding value for *STRUC*. This can be attributed to the following reason. Substantial changes in the pattern of method calls (which are captured in sequence diagrams) for software of the same family do not necessarily translate to significant changes in UML structure diagrams such as class diagrams.

All the multi-view similarity methods matched the results of *STRUC* in terms of correlation with reuse effort and MAP, but required more retrieval time than *STRUC*. *SIMULT0.75* had the overall best MAP of 98.50%, while *COMP0.75* produced the best correlation coefficient of 84.05%, which is marginally higher than the corresponding value of 82.77% obtained by using *STRUC*.

Of all the multi-view approaches, multi-view by cascading required the lowest retrieval time because functional similarity assessment was done using only the mapping of sequence diagrams, since class mappings were carried over from the preceding stage (i.e., structural similarity assessment stage). On the other hand, it can be observed from Figure 10.12 that among all the methods, the simultaneous search architecture required the largest amount of time for retrieval.

Since *STRUC* performed better than *FXN* on all the evaluated parameters, it is reasonable to expect that assigning a higher weight to structural similarity measure than functional similarity measure (i.e., $w_1 > w_2$) would improve the performance of the multi-view similarity assessment. We investigated this possibility by using three values of w_1 (0.25, 0.50 and 0.75). In majority of the cases, using a higher weight for w_1 resulted in the best overall values for MAP and correlation coefficient. In a few cases, using equal weights for w_1 and w_2 produced the best values of MAP and correlation coefficient. It is noteworthy that setting $w_1 = 0.25$ produced the worst result for each of the multi-view architectures.

10.6.4 Conclusion

Experiments showed that the multi-view approaches produced better results compared to when only functional similarity assessment was used. In some cases, multi-view approaches resulted in slightly better values of MAP and correlation with predicted reuse effort compared to structural similarity assessment, but were slower than structural similarity assessment.

Each of the multi-view approaches had its own strength: the composition approach requires the least computational time of the three approaches; the highest MAP was obtained using the simultaneous search approach; and the view composition approach gave the best correlation with predicted reuse effort.

10.7 Experiment 5

This section describes an experiment to evaluate the different multi-view approaches. Unlike Experiment 4, this experiment was carried out using a dataset containing class diagrams, sequence diagrams and state machine diagrams.

10.7.1 Experimental Data

Four queries containing class diagrams, sequence diagrams and state machine diagrams were used for this experiment. The queries were obtained from course projects for undergraduate and graduate students of software engineering. The number of different UML diagrams contained in each query is given in Table 10.10

Table 10.10: Properties of the four queries

	Q_1	Q_2	Q_3	Q_4
# class diagrams	1	1	1	1
# sequence diagrams	1	12	5	3
# state machine diagrams	1	2	1	3

Six repositories were formed by randomly making different degrees of changes in the query diagrams. Class diagram relationships were changed from one type to another; for example, a composition relationship may be randomly changed to a generalization relationship. The source and destination objects for sequence diagram messages were also randomly modified. Furthermore, the types of edges in state machine diagrams (i.e., transition edges, hierarchical edges, beginning edges and ending edges) were randomly changed.

Table 10.11 shows the number of projects in the different repositories. In the first repository, no change is made to the queries, so the repository simply consists of the four queries. Each repository project is only relevant to the corresponding query. For the remaining five repositories, 25 projects are randomly created per query in the manner described in the previous paragraph. For example, in order to populate the second repository, for each query, 25 repository projects are formed by randomly making changes with a probability of 0.2 in each diagram in the query. Each repository project is only relevant to the query from which it was created.

Table 10.11: Number of projects in the repositories

Probability of Change (%)	0	20	40	60	80	100
# projects in repository	4	100	100	100	100	100

10.7.2 Results

The parameters used to run the multi-view experiments are shown in Table 10.12. The mean of MAP as well as average time required to search a repository of 100 projects for the different methods are presented in Figure 10.13 and Figure 10.14, respectively. Because the simultaneous search multi-view architecture produced the best overall MAP values (see Figure 10.13), we compared 2-view simultaneous search and 3-view simultaneous search architectures. Each 2-view architecture had one of the three weights set to zero, while the other two weights were each set to 0.5. For example, $w_1 = 0.5$, $w_2 = 0.5$ and $w_3 = 0$ for *STUCT+FXN*. Figure 10.15 shows that 3-view simultaneous search performed better than any of the 2-view simultaneous search architectures.

Table 10.12: Parameters for Experiment 5

Parameter Description	Value
Size of population	50
Maximum number of generations	1000
Number of generations to terminate algorithm if fitness value does not improve	20
Number of individuals from initial population produced using Munkres' algorithm	3
Probability of mutating class genes	0.01
Probability of mutating sequence diagram genes	0.01
Probability of mutating state machine diagram genes	0.1
Number of repetitions	5
α in Eq. (5.1)	0.05
γ in Eq. (6.3)	0.15
λ in Eq. (7.1)	0.05
Θ in Eq. (7.2)	0.15
(w_1, w_2, w_3) in equations (8.1) and (8.2)	(0.33, 0.33, 0.33)

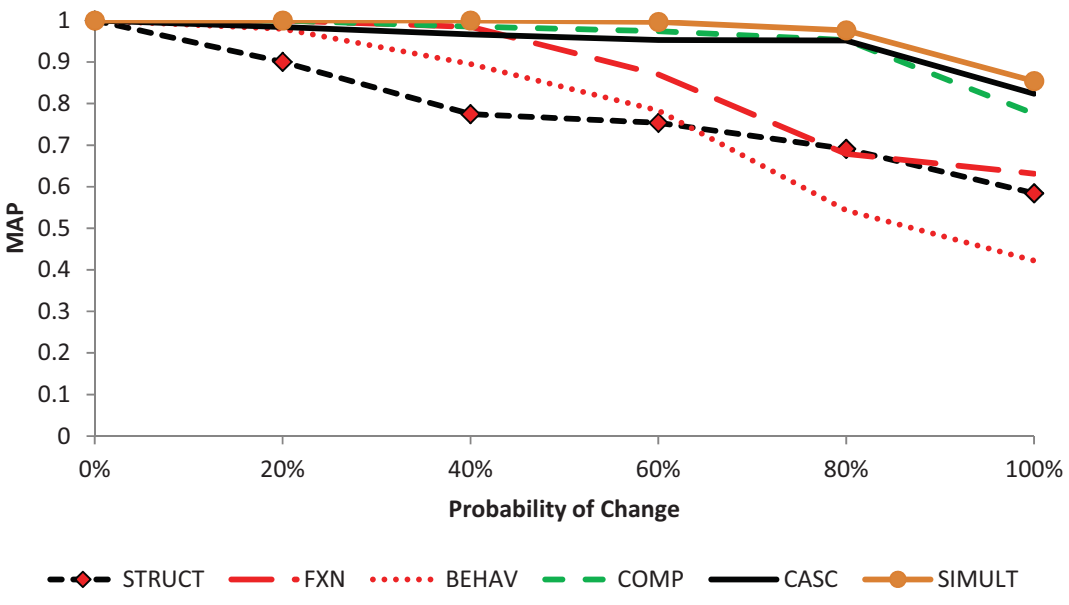


Figure 10.13: Mean MAP for single view and 3-view similarity assessment

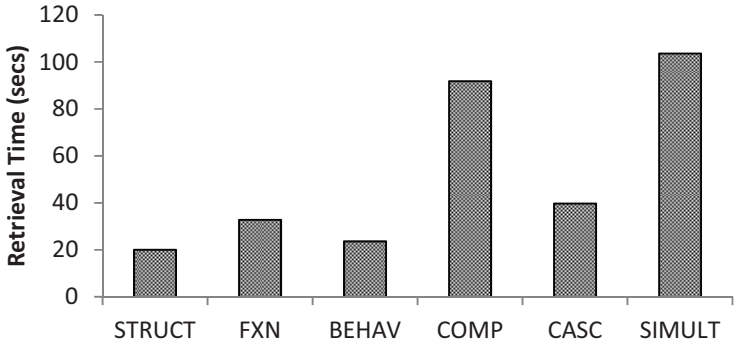


Figure 10.14: Mean retrieval time for different methods

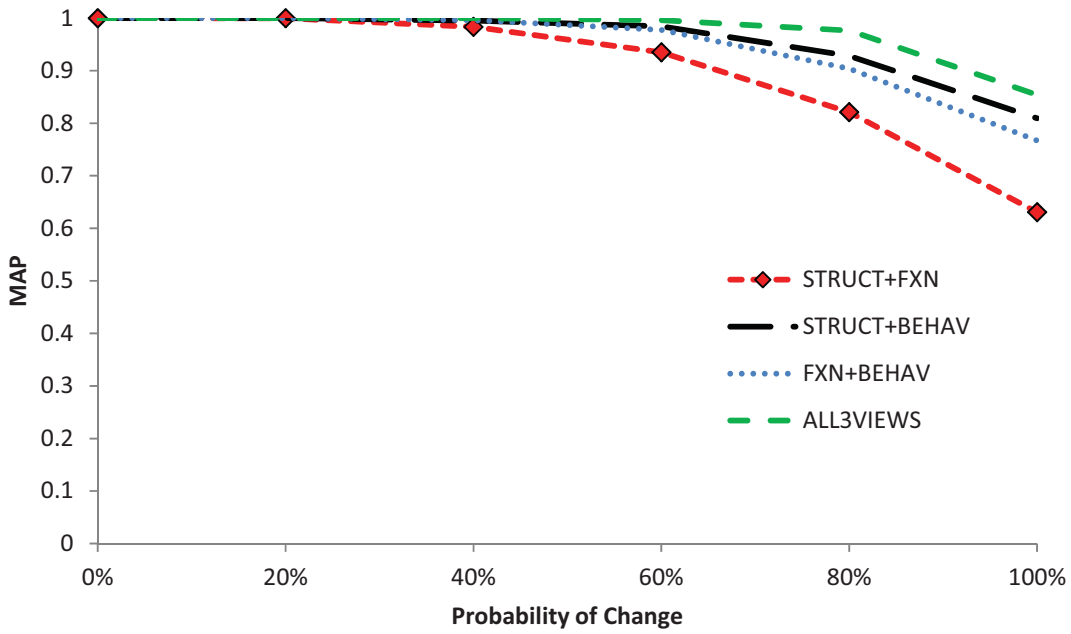


Figure 10.15: Mean MAP for 2-view and 3-view simultaneous search architecture

10.7.3 Discussion of Results

It can be observed from Figure 10.13 that 3-view similarity assessment results in better MAP than single view similarity assessment, especially when the query and relevant

repository projects differ significantly. Figure 10.14 shows that single view similarity assessment requires less retrieval time than multi-view similarity assessment.

As was the case in the Experiment 4, cascading required the least retrieval time among the multi-view architectures, while the simultaneous search architecture produced the best overall mean MAP.

Figure 10.15 shows that multi-view similarity assessment using all the three views (structural, functional and behavioral views) results in better mean MAP than any combination of two views. It is noteworthy that a combination of structural and functional views produces the poorest MAP of all the 2-view similarity scores. This explains why, in the Experiment 4, a combination of structural and functional views performed slightly better than when single view similarity assessment was carried out.

10.7.4 Conclusion

The results of this experiment corroborate those obtained in Experiment 4. In conclusion, single view and 2-view similarity assessment result in lower MAP than 3-view similarity assessment.

10.8 Experiment 6

This section describes an experiment to evaluate the different multi-view approaches. Like in Experiment 5, this experiment was carried out using a dataset containing class diagrams, sequence diagrams and state machine diagrams.

10.8.1 Experimental Data

Eight queries containing class diagrams, sequence diagrams and state machine diagrams were used for this experiment. The queries were obtained from course projects for undergraduate and graduate students of software engineering. The number of different UML diagrams contained in each query is given in Table 10.13. The last four queries ($Q_5 \ldots Q_8$) were UML diagrams from an undergraduate course project in which groups of students were asked to independently design an estate management system for apartment/house rental listing. The first four queries were for a microprocessor simulator, ATM system, online diary and car rental system.

Eight repository models $R_1 \ldots R_8$ were formed by using each of the query models in turn (i.e, $R_i = Q_i$, $1 \leq i \leq 8$). The similarity between each query model and every repository model was determined. Figure 10.16 shows which repository models are relevant to each query. The grey colored cells indicate relevance. Note that the last four repository models are relevant to the last four queries, since four groups of students were asked to design the same system – an estate management system.

Table 10.13: Properties of the eight queries

	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8
# class diagrams	1	1	1	1	1	1	1	1
# sequence diagrams	1	12	5	3	17	21	8	8
# state machine diagrams	1	2	1	3	4	5	4	3

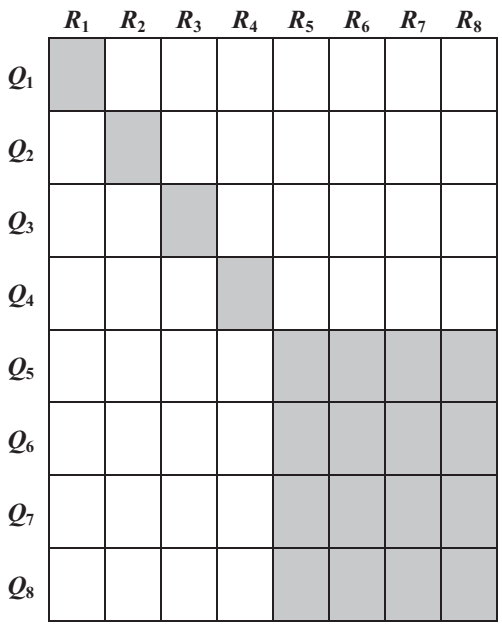


Figure 10.16: Relevance of query and repository models

10.8.2 Results and Discussion

The parameters used to run the experiments are the same as those presented in Table 10.12. The mean of MAP as well as average time required to search the repository for the different methods are presented in Figure 10.17 and Figure 10.18, respectively.

As expected, cascading required the least retrieval time among all the multi-view architectures. Like in the previous two experiments, simultaneous search resulted in the best MAP among the multi-view architectures. However, functional similarity assessment produced slightly better MAP than multi-view similarity assessment with all three views. Notwithstanding, all the methods produced very high values of MAP; behavioral similarity assessment produced the lowest MAP value of 87.71%.

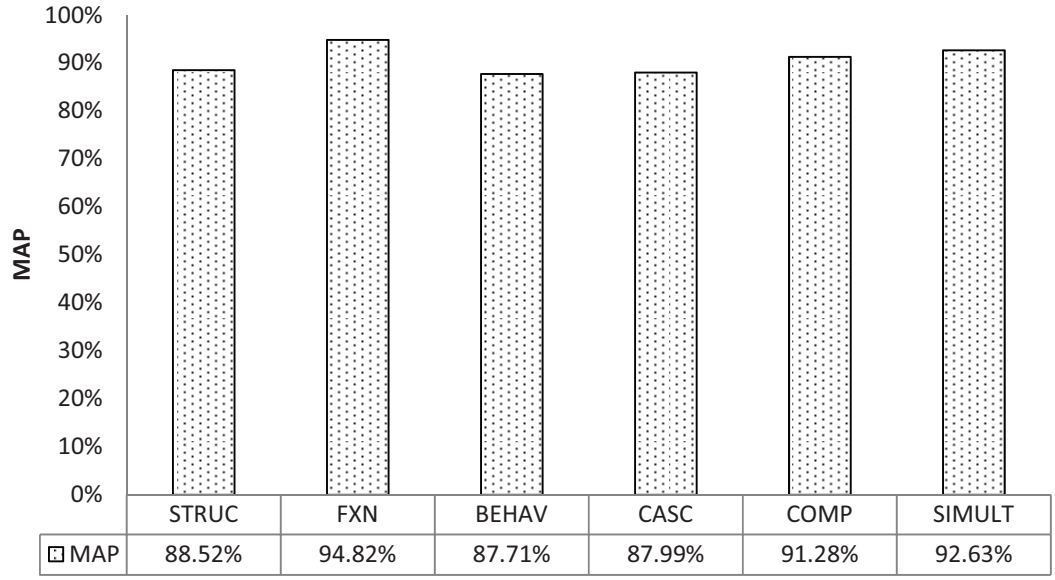


Figure 10.17: Mean MAP for single view and 3-view similarity assessment

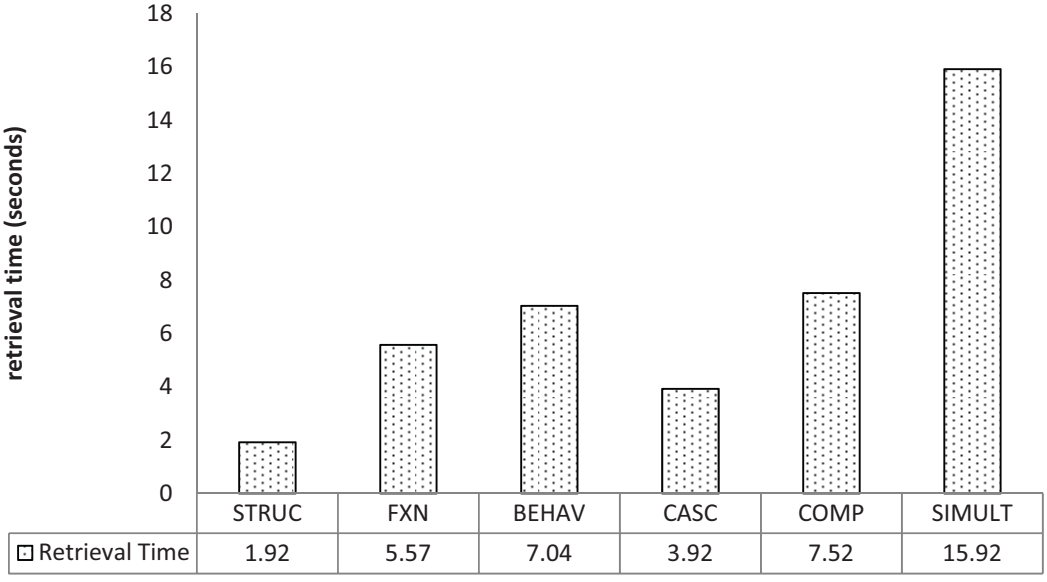


Figure 10.18: Mean retrieval time for different methods

10.8.3 Conclusion

The results of this experiment corroborate those obtained in Experiments 4 and 5 in several aspects: among the multi-view architectures, simultaneous search and cascading resulted in the highest MAP and least retrieval time, respectively.

The results of this experiment differed from those obtained in Experiments 4 and 5 in one aspect: functional similarity assessment produced the best overall MAP. However, since the repository contained very few projects (eight projects), the superior performance of functional similarity assessment over multi-view similarity assessment does not necessarily extend to cases when there are many projects in the repository.

10.9 Experiment 7

This section describes a set of experiments to evaluate the impact of pre-filtering.

10.9.1 Experimental Data

The dataset of section 10.3.1 was used in this experiment.

10.9.2 Results

We began the series of experiments by determining the performance of the pre-filtering stage without a retrieval stage. Table 10.14 shows the retrieval time as well as correlation between pre-filtering similarity scores and reuse effort when pre-filtering is considered as a stand-alone retrieval stage.

In another experiment, we measured the MAP as the number of projects returned by the pre-filtering stage is varied. The horizontal axis of Figure 10.19 shows that the number of projects returned after pre-filtering is varied from 5 to 30. The vertical axis shows the

MAP. Note that when computing MAP using Eq. (10.4), the precision for relevant documents that were not shortlisted is zero.

The final experiment studied the effect of pre-filtering on MAP and retrieval time for each method listed in Table 10.8. Figure 10.20 and Figure 10.21 show the MAP and retrieval time, respectively, for the different methods with and without pre-filtering. In the experiment, 10 projects were shortlisted at the end of the pre-filtering stage.

Table 10.14: Performance of pre-filtering stage without a retrieval stage

Correlation with reuse effort	**time to search repository (milliseconds)**
Correlation Coefficient = 0.6130 Significance Level = 2.56×10^{-49}	3.99

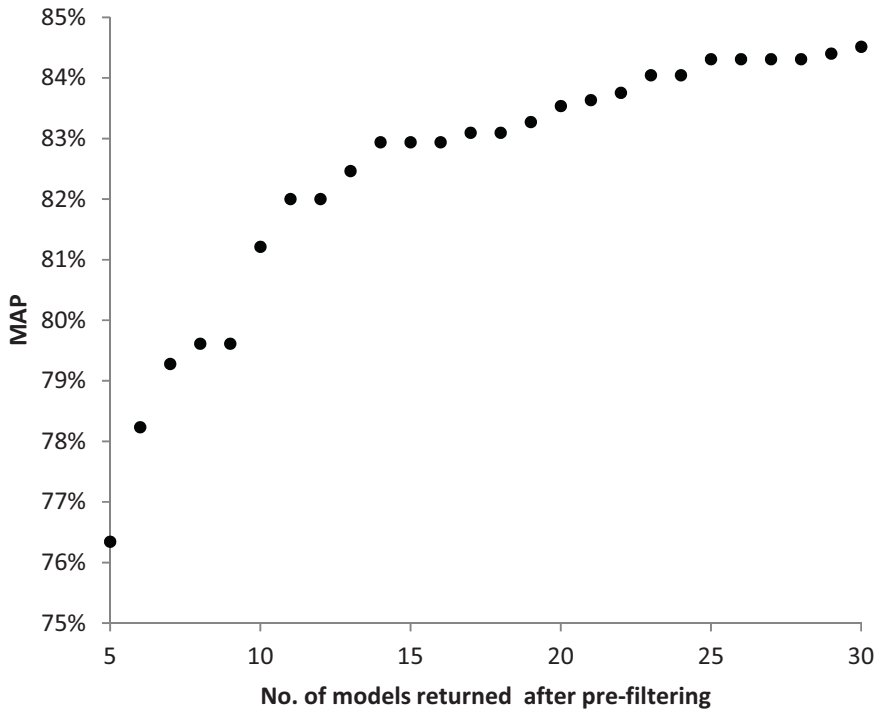
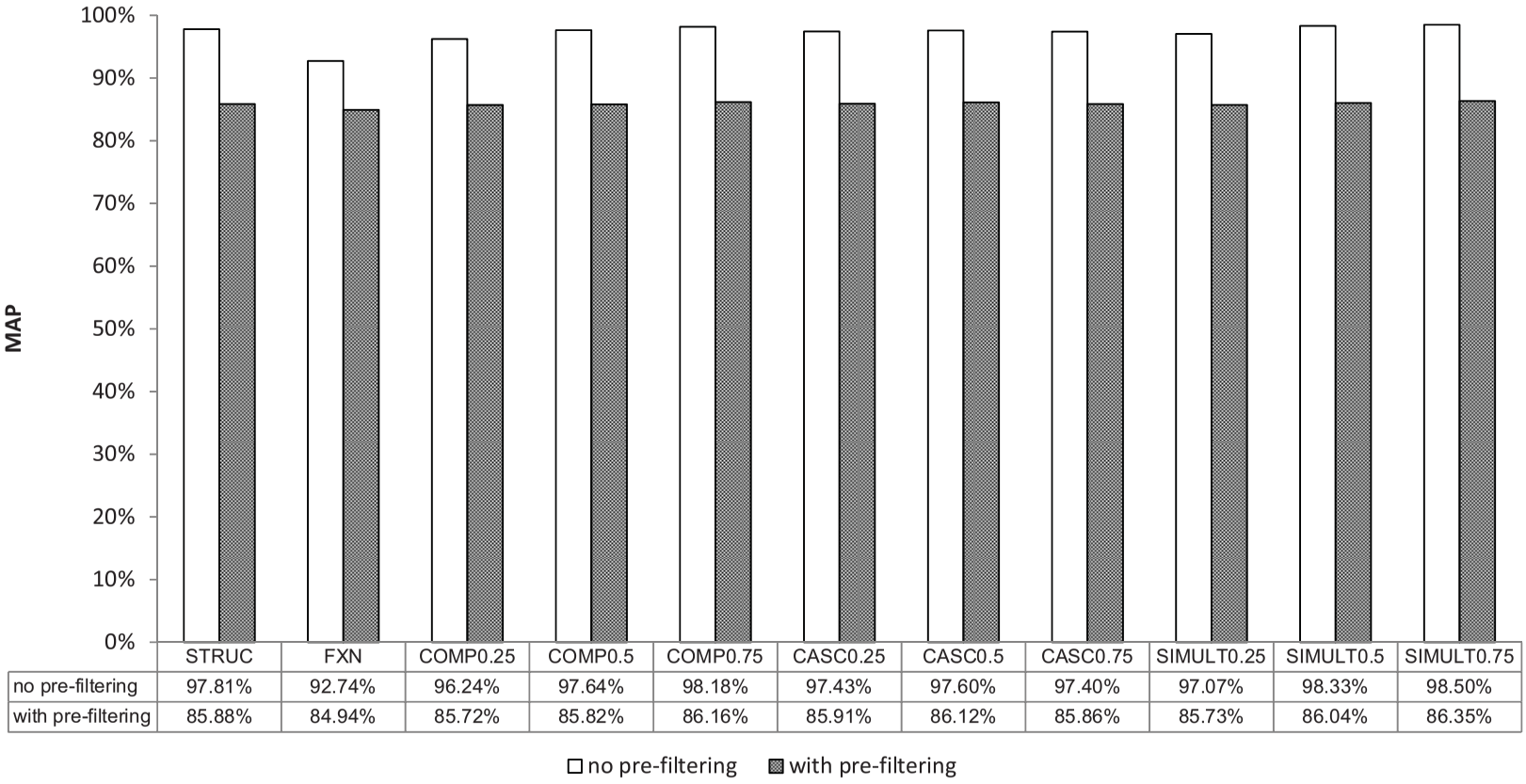
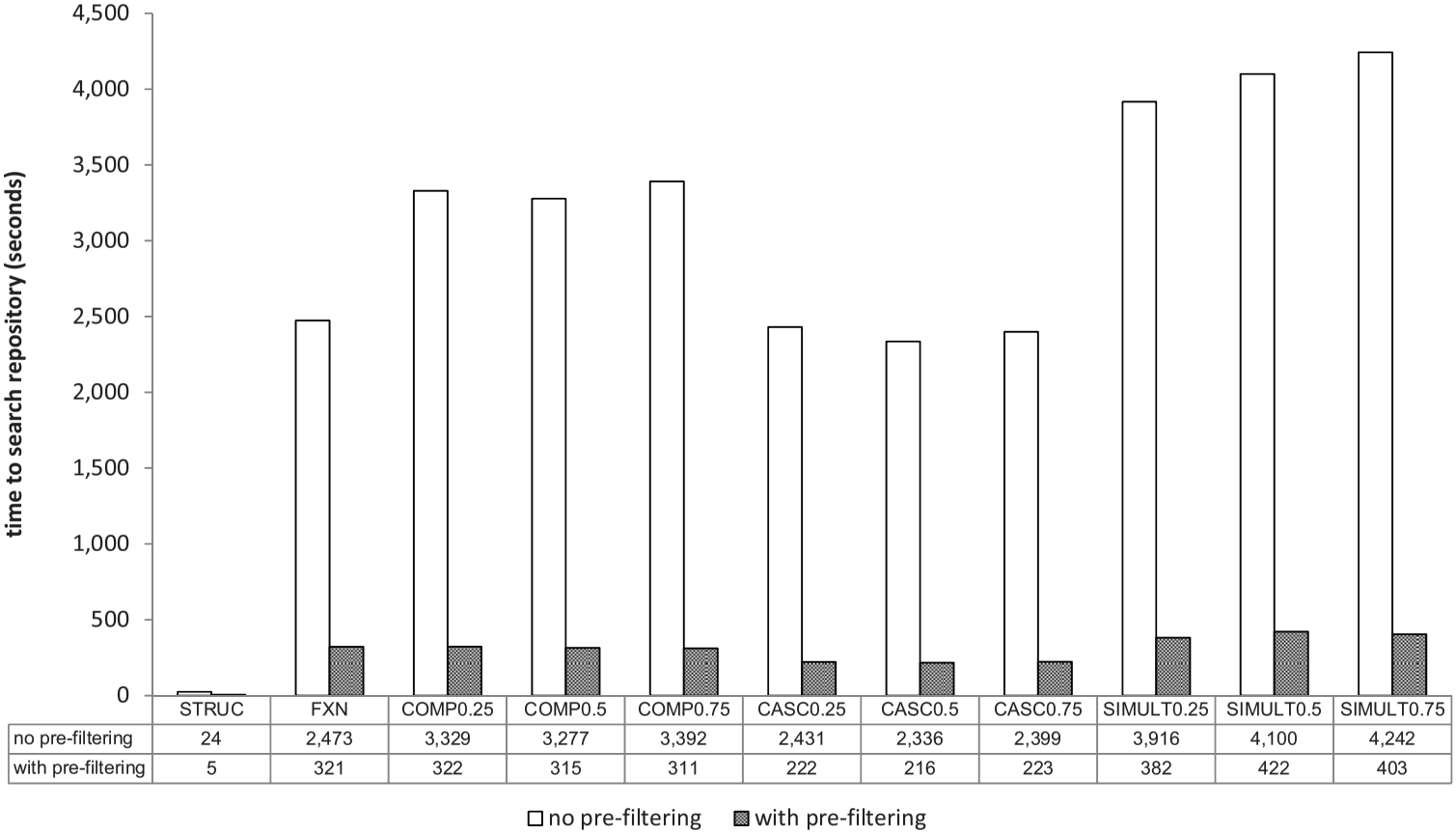


Figure 10.19: Relationship between MAP and number of projects selected after pre-filtering



	STRUC	FXN	COMP0.25	COMP0.5	COMP0.75	CASC0.25	CASC0.5	CASC0.75	SIMULT0.25	SIMULT0.5	SIMULT0.75
no pre-filtering	97.81%	92.74%	96.24%	97.64%	98.18%	97.43%	97.60%	97.40%	97.07%	98.33%	98.50%
with pre-filtering	85.88%	84.94%	85.72%	85.82%	86.16%	85.91%	86.12%	85.86%	85.73%	86.04%	86.35%

Figure 10.20: Effect of pre-filtering on MAP



	STRUC	FXN	COMP0.25	COMP0.5	COMP0.75	CASC0.25	CASC0.5	CASC0.75	SIMULT0.25	SIMULT0.5	SIMULT0.75
no pre-filtering	24	2,473	3,329	3,277	3,392	2,431	2,336	2,399	3,916	4,100	4,242
with pre-filtering	5	321	322	315	311	222	216	223	382	422	403

Figure 10.21: Effect of pre-filtering on search time

10.9.3 Discussion of Results

As can be inferred from Table 10.14 and Figure 10.19, our pre-filtering approach meets its objective; it is computationally inexpensive. Moreover, it returns many of the relevant repository projects in response to the queries, since the MAP is 76.34% when only five projects are shortlisted (see Figure 10.19). However, the correlation between the pre-filtering similarity score and predicted reuse effort is only 61.30%. This does not matter much, since the actual similarity scores between query and repository projects will be determined in the retrieval stage.

Figure 10.20 shows that pre-filtering leads to a reduction in MAP for each method. This decrease in MAP is expected since pre-filtering inadvertently omits some relevant repository projects as a result of its superficial comparison, which is based on only 16 metrics. Pre-filtering caused a drop in MAP of between 7% and 13% for different methods, resulting in the MAPs lying between 84% and 87%.

It can be observed from Figure 10.21 that pre-filtering also led to a sharp decline in retrieval time. Except for *STRUC* whose retrieval time dropped from 24.06 to 5.20 seconds, the retrieval time of all other methods improved by a factor of approximately 10. Consequently, the retrieval time for the various methods was reduced to between 5.20 and 421.58 seconds.

10.9.4 Conclusion

Experimental results show that in all but one case, a speedup factor of approximately 10 was gained when pre-filtering was used. Pre-filtering led to a reduction in MAP and low correlation with reuse effort. Reduction in MAP is expected since pre-filtering mistakenly

filters out relevant repository projects owing to its use of a shallow approach for selecting projects' based on supposed similarity. It does not matter that there is a low degree of correlation between pre-filtering scores and estimated reuse effort since pre-filtering is expected to be followed by the retrieval stage, which has been shown to produce strong degree of correlation with reuse effort in previous experiments.

10.10 Chapter Conclusion

The results of experiments carried out in this chapter show that our similarity assessment techniques perform very well in terms of retrieval quality (MAP) and correlation of similarity values with reuse effort. In all but one experiment, multi-view similarity assessment produced better results than single view similarity assessment. (Shallow) structural similarity assessment required the least retrieval time among all the similarity assessment techniques. Pre-filtering has proved to be a useful technique for minimizing retrieval time, especially when the repository contains many models.

Table 10.15 summarizes our recommendations. When the repository is not large and the retrieval results are required immediately, it is recommended to use shallow structural similarity assessment without pre-filtering, since it is very fast and results in very good MAP and correlation with reuse effort. However, if the repository is small, and the reuser can tolerate some delay, it is recommended to use multi-view similarity assessment without pre-filtering, since it has been shown to result in better retrieval quality compared to single view similarity assessment in most cases.

For large repositories, we suggest using pre-filtering prior to full-fledged retrieval. During retrieval, shallow structural similarity assessment is utilized if the retrieval must

be done instantaneously, whereas multi-view similarity assessment is carried out if the reuser can endure some delay during retrieval.

Table 10.15: Recommended similarity assessment techniques for different scenarios

	Instantaneous Response	**Delayed Response**
Small repository	Shallow structural similarity assessment	Multi-view similarity assessment
Large repository	Pre-filtering followed by shallow structural similarity assessment	Pre-filtering followed by Multi-view similarity assessment

Chapter 11

Conclusions and Directions for Further Work

In this chapter, we summarize our work, describe various threats to the validity of our work, and provide directions for future research.

11.1 Summary

In order to exploit the gains of early-stage software reuse, this dissertation has discussed the retrieval of software based on the degree of similarity of their UML models. Two key problems were addressed:

1. Software systems are usually described using models that show the systems from multiple perspectives, which may result in inconsistencies if similarity assessment is not properly carried out.
2. Models often contain many entities, thus exhaustively pairing this entities during similarity assessment is computationally demanding

In order to tackle these problems, similarity assessment was construed as involving two sub problems: entity matching and similarity scoring. The former problem was concerned with establishing a mapping between entities of the same type, while the latter problem

required the use of similarity measures to compute the degree of similarity of two models based on the established mappings.

Two heuristic search algorithms were used to perform matching: GA and CSA. The matching process was enhanced by the use of similarity matrices which were computed from the properties of entities that needed to be matched. The similarity matrices were used during population initialization (for GA and CSA) and crossover (for GA). Both search algorithms resulted in more or less the same retrieval performance, even though GA was much faster than CSA. Thus, it is recommended to use GA, rather than CSA for matching.

Shallow structural similarity assessment was carried out by measuring the degree of structural similarity of graph representation of class diagrams. The degree of dissimilarity between class diagram relationships was obtained from a lookup table whose information was gathered from UML experts. Out technique produced excellent results in a reasonable time: high MAP; and a strong correlation between similarity values and estimated reuse effort. Deep structural similarity assessment compared the constituent parts of classifiers in order to compute the similarity between class diagrams. It produced high MAP and good correlation with reuse effort, but required more time than shallow similarity assessment.

Two methods were used to perform functional similarity assessment: the first method is useful for comparing two sequence diagrams, and involves a comparison of the graph representation of sequence diagrams; in the second method which is suitable for comparing sets of sequence diagrams, we introduced a novel technique which involved

finding the longest common subsequence of matching messages in sequence diagrams. Experimental results for both methods showed that our methods are very effective at retrieving sequence diagrams.

Behavioral similarity assessment was carried out by computing the similarity of the UML state machine diagrams describing the behavior of objects (i.e., class instances) in a system. State machine diagrams were converted to graphs which were then compared. Experimental results indicate that our method of comparing software using their state machine diagrams lead to the retrieval of similar software from the repository.

Three architectures for consistent multi-view similarity assessment were presented in this dissertation. In all these architectures, an overall multi-view similarity score was obtained by aggregating the structural, behavioral and functional similarity scores. Experimental results showed that in many cases multi-view similarity assessment performed slightly better than single view similarity assessment, but the former required much more time than the latter.

In order to reduce the retrieval time to a reasonable value, a quick way of shortlisting some potentially similar projects to the query was devised. Pre-filtering led to a drastic decrease in retrieval time, even though it led to a little degradation of the MAP.

It is recommended to use shallow structural similarity assessment when retrieval results are required instantaneously, whereas multi-view similarity assessment should be used when the reuser does not object to some delay during retrieval. However, when the repository contains many projects, a pre-filtering stage should precede the retrieval stage.

11.2 Threats to Validity

This section discusses different threats to the validity of our study and results.

11.2.1 Construct Validity

Construct validity is concerned with whether the studied parameters are relevant to the research questions [75]. In the case of our study, high construct validity means that we used the right measure to evaluate the effectiveness of our retrieval techniques. The three measures used to evaluate our retrieval techniques were retrieval time, MAP and correlation with reuse effort. We believe that this study has a low threat to construct validity for the following reasons.

MAP is widely used for evaluating ranked retrieval systems. In fact, it is the most standard measure used in the Text REtrieval Conference (TREC), a prestigious information retrieval conference [17].

The retrieval time of the different techniques was measured, because a retrieval technique with excellent MAP may never be used if it requires unacceptably long time to execute.

Even though the formula we applied for estimating reuse effort was originally used by Basili et al. [70] for predicting the effort for maintaining releases of software products, maintenance of software releases and software reuse both require modifying existing software to obtain new ones. Moreover, we measured correlation between reuse effort and similarity scores only for experiments that used the dataset of Section 10.3.1, which comprises reverse engineered UML diagrams for different software releases.

11.2.2 Internal Validity

Internal validity ensures that if a relationship is observed between treatment and outcome, it must be guaranteed that it is a causal relationship, not as a result of factors upon which the researcher has no control [76].

It can be argued that intuitively, different releases of the same software are more similar to themselves than to other software, hence the high values of MAP obtained by our techniques can be obtained using simpler techniques. This is corroborated by Figure 10.19 which shows that pre-filtering alone produces very high values of MAP. Nevertheless, our retrieval techniques result in MAP values that are almost 15% higher than those produced by pre-filtering alone. Furthermore, a comparison of Figure 10.11 and Table 10.14 shows that our retrieval techniques produce much better correlation with reuse effort than pre-filtering alone.

11.2.3 External Validity

External validity refers to the applicability of experimental results outside the scope of the study [75]. We minimized the threat to external validity by selecting software of different sizes and belonging to different domains. The UML diagrams (see Table 10.1) used for several of the experiments have 11—66 classifiers and 15—254 sequence diagrams containing 172—9291 messages. Similarly, other datasets contained UML diagrams of various sizes which often belonged to several domains.

11.3 Limitations and Future Work

This section describes limitations of the work reported in this dissertation, and states how the work can be extended.

1. Shallow similarity assessment of class diagrams was carried out in Section 5.1 by considering only the differences in the types of class diagrams relationships. The work can be extended by taking into account the roles and multiplicity of associations.
2. In Section 5.1 shallow similarity assessment compared the structural information contained in class diagrams, whereas deep similarity assessment in Section 5.2 took into account the lexical similarity of the inner parts of classifiers. An aggregation of deep and shallow similarity values may produce better similarity values (in terms of MAP and correlation with reuse effort) than either of the two similarity values on its own.
3. None of the functional similarity assessment techniques described in Chapter 6 took into account combined fragments in sequence diagrams such as alternatives, options and loops. Future work could account for combined fragments contained in sequence diagrams.
4. In Sections 6.2 and 6.3, the messages in two sequence diagrams were considered as matched if their source and destination classes were mapped. This may result in the matching of different messages belonging to the same class. A more fine-grained – but computationally expensive – approach will involve mapping the methods of all mapped classes in order to determine the longest common subsequence of mapped messages.
5. The functional similarity assessment described in Sections 6.2 and 6.3 can be better adapted to improve reuse of software within the same domain by considering classifier names and message names alongside the length of LCSMM.

6. In Chapter 7, behavioral similarity assessment did not take into account the events, guard conditions and actions of transitions, as well as the names of states in state machine diagrams. Consideration of all these pieces of information may lead to better MAP.
7. In Chapter 8, multi-view similarity value was aggregated from structural, behavioral and functional similarity scores. Thus, the search for an optimal overall score is multi-objective. Rather than relying on weights to determine the relative importance of structural, behavioral and functional similarity scores, multi-objective versions of heuristic search algorithms can be used to offer the reuser a set of solutions that represent a good tradeoff between the objectives.
8. We showed that similarity values from our methods had strong degree of correlation with estimated reuse effort. More research effort needs to be directed in this area, with a view to developing techniques for accurately predicting reuse effort by simply comparing early-stage models.
9. Currently, the number of projects returned after pre-filtering has been left to the reuser to determine. More research is needed to automatically determine the fraction of repository projects to be returned after pre-filtering. It is important to note that returning a large fraction of repository projects after pre-filtering defeats the aim of pre-filtering. On the other hand, returning a small fraction of repository projects may lead to a significant decrease in MAP, since many relevant projects may not be shortlisted at the end of pre-filtering.

BIBLIOGRAPHY

[1] W. B. Frakes and K. Kang, "Software Reuse Research: Status and Future," *IEEE Trans. Softw. Eng.,* vol. 31, pp. 529-536, 2005.

[2] I. Sommerville, *Software Engineering*, 7th ed.: Pearson Addison Wesley, 2004.

[3] S. R. Schach, *Object-Oriented Software Engineering*. New York: McGraw-Hill, 2008.

[4] A. Prasad and E. K. Park, "Reuse system: An artificial intelligence - based approach," *Journal of Systems and Software,* vol. 27, pp. 207-221, 1994.

[5] R. A. Rufai, "New Structural Similarity Metrics for UML Models." vol. M.Sc. Thesis: King Fahd University of Petroleum and Minerals, Saudi Arabia, 2003.

[6] J. L. Cybulski, R. D. B. Neal, A. Kram, and J. C. Allen, "Reuse of early life-cycle artifacts: workproducts, methods and tools," *Ann. Softw. Eng.,* vol. 5, pp. 227-251, 1998.

[7] M. Ahmed, "Towards the Development of Integrated Reuse Environments for UML Artifacts," in *ICSEA 2011, The Sixth International Conference on Software Engineering Advances*, 2011, pp. 426-431.

[8] J. Iivari, "Object-orientation as structural, functional and behavioural modelling: a comparison of six methods for object-oriented analysis," *Information and Software Technology,* vol. 37, pp. 155-163, 1995.

[9] G. Kotonya and I. Sommerville, *Requirements Engineering: Processes and Techniques*: John Wiley and Sons, 1998.

[10] J. R. Rumbaugh, M. R. Blaha, W. Lorensen, F. Eddy, and W. Premerlani, *Object-oriented modeling and design*, First ed.: Prentice Hall, 1991.

[11] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, 2nd ed.: Addison-Wesley, 2005.

[12] O. Edelstein, A. Yaeli, and G. Zodik, "eColabra: An Enterprise Collaboration & Reuse Environment," in *Fourth Int'l Workshop (NGITS'99)*, 1999, pp. 229-236.

[13] OMG, "Unified Modeling Language Superstructure Specification V2.4.1," 2011.

[14] I. Jacobson, *Object Oriented Software Engineering: A Use Case Driven Approach*: Addison-Wesley, 1992.

[15] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*: Addison Wesley Longman, 1999.

[16] P. Roques, *UML in Practice: The Art of Modeling Software Systems Demonstrated through Worked Examples and Solutions*: Wiley, 2004.

[17] C. D. Manning, P. Raghavan, and H. Schtze, *Introduction to Information Retrieval*: Cambridge University Press, 2008.

[18] K. Wolter, T. Kreb, and L. Hotz, "Determining Similarity of Model-based and Descriptive Requirements by Combining Different Similarity Measures," in *Proceedings of 2nd International Workshop on Model Reuse Strategies*, 2008.

[19] M. C. Blok and J. L. Cybulski, "Reusing UML Specifications in a Constrained Application Domain," in *Proceedings of the Fifth Asia Pacific Software Engineering Conference*: IEEE Computer Society, 1998.

[20] T. A. Alspaugh, A. I. Ant, T. Barnes, and B. W. Mott, "An Integrated Scenario Management Strategy," in *Proceedings of the 4th IEEE International Symposium on Requirements Engineering*: IEEE Computer Society, 1999, pp. 142-149.

[21] F. M. Ali and W. Du, "Toward reuse of object-oriented software design models," *Information and Software Technology,* vol. 46, pp. 499 - 517, 2004.

[22] P. Gomes, F. C. Pereira, P. Carreiro, P. Paiva, N. Seco, J. L. Ferreira, and C. Bento, "Case-Based Adaptation for UML Diagram Reuse," in *KES*: Springer, 2004.

[23] P. Gomes, F. C. Pereira, P. Paiva, N. Seco, P. Carreiro, J. L. Ferreira, and C. Bento, "Case Retrieval of Software Designs using WordNet," in *European Conference on Artificial Intelligence (ECAI 02)*, 2002, pp. 245-249.

[24] P. Gomes, F. C. Pereira, P. Paiva, N. Seco, P. Carreiro, J. L. Ferreira, and C. Bento, "Experiments on Case-Based Retrieval of Software Designs," in *Proceedings of the 6th European Conference on Advances in Case-Based Reasoning*: Springer-Verlag, 2002, pp. 118-132.

[25] P. Gomes, F. C. Pereira, P. Paiva, N. Seco, P. Carreiro, J. L. Ferreira, and C. Bento, "Case-Based Reuse of UML Diagrams," in *The Fifteen International Conference on Software Engineering and Knowledge Engineering (SEKE 2003)*, 2003, pp. 335-339.

[26] P. Gomes, F. C. Pereira, P. Paiva, N. Seco, P. Carreiro, J. L. Ferreira, and C. Bento, "Using WordNet for case-based retrieval of UML models," *AI Commun.,* vol. 17, pp. 13-23, 2004.

[27] W. N. Robinson and H. G. Woo, "Finding Reusable UML Sequence Diagrams Automatically," *IEEE Softw.,* vol. 21, pp. 60-67, 2004.

[28] I. Jonyer, D. J. Cook, and L. B. Holder, "Graph-based hierarchical conceptual clustering," *J. Mach. Learn. Res.,* vol. 2, pp. 19-43, 2002.

[29] J. Llorens, J. M. Fuentes, and J. Morato, "UML retrieval and reuse using XMI," in *IASTED Conf. on Software Engineering*, 2004, pp. 740-746.

[30] S. Channarukul, S. Charoenvikrom, and J. Daengdej, "Case-based reasoning for software design reuse," in *Aerospace Conference, 2005 IEEE*, 2005, pp. 4296-4305.

[31] A. Ahmed, "Functional similarity metric for UML models," King Fahd University of Petroleum and Minerals, 2006.

[32] H. B. Khalifa, O. Khayati, and H. H. B. Ghezala, "A Behavioral and Structural Components Retrieval Technique for Software Reuse," in *Proceedings of the 2008 Advanced Software Engineering and Its Applications*: IEEE Computer Society, 2008, pp. 134-137.

[33] D. Bildhauer, T. Horn, and J. Ebert, "Similarity-driven software reuse," in *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*: IEEE Computer Society, 2009, pp. 31-36.

[34] G. Engels, B. Opdyke, D. Schmidt, F. Weil, M. ÅšmiaÅ‚ek, J. Bojarski, W. Nowakowski, A. Ambroziewicz, and T. Straszak, "Complementary Use Case Scenario Representations Based on Domain Vocabularies," in *Model Driven Engineering Languages and Systems*. vol. 4735: Springer Berlin Heidelberg, 2007, pp. 544-558.

[35] M. Smialek, A. Kalnins, E. Kalnina, A. Ambroziewicz, T. Straszak, and K. Wolter, "Comprehensive System for Systematic Case-Driven Software Reuse," in *Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science*: Springer-Verlag, 2010, pp. 697-708.

[36] Y. Kotb, "Applying the Textual Entailment Approach to Automatic Reusable Software," in *The 7th International Conference on Informatics and Systems (INFOS)*, 2010, pp. 1-6.

[37] W.-J. Park and D.-H. Bae, "A two-stage framework for UML specification matching," *Inf. Softw. Technol.,* vol. 53, pp. 230-244, 2010.

[38] B. Bonilla-Morales, S. Crespo, and C. Clunie, "Reuse of Use Case Diagrams: An Approach Based on Ontologies and Semantic Web Technologies," *International Journal of Computer Science Issues,* vol. 9, pp. 24-29, 2012.

[39] K. Robles, A. Fraga, J. Morato, and J. Llorens, "Towards an ontology-based retrieval of UML Class Diagrams," *Information and Software Technology,* vol. 54, pp. 72-86, 2012.

[40] H. O. Salami and M. Ahmed, "A Framework for Class Diagram Retrieval Using Genetic Algorithm," in *The 24th International Conference on Software Engineering and Knowledge Engineering (SEKE 2012)*: Knowledge Systems Institute Graduate School, 2012, pp. 737-740.

[41] H. O. Salami and M. Ahmed, "Class Diagram Retrieval Using Genetic Algorithm," in *12th International Conference on Machine Learning and Applications* Miami, Florida, 2013, pp. 96-101.

[42] W. K. G. Assuncao and S. R. Vergilio, "Class Diagram Retrieval with Particle Swarm Optimization," in *The 25th International Conference on Software Engineering and Knowledge Engineering (SEKE 2013)*, 2013, pp. 632 - 637.

[43] J. L. Cybulski, "Introduction to Software Reuse," The University of Melbourne, Australia 1996.

[44] M. Stephan and J. R. Cordy, "A Survey of Methods and Applications of Model Comparison," Queen's University, Canada 2012.

[45] C. Bunse and C. Atkinson, "The normal object form: bridging the gap from models to code," in *Proceedings of the 2nd international conference on The unified modeling language: beyond the standard* Fort Collins, CO, USA: Springer-Verlag, 1999.

[46] R. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," in *2007 Future of Software Engineering*: IEEE Computer Society, 2007.

[47] W. Harrison, C. Barton, and M. Raghavachari, "Mapping UML designs to Java," in *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* Minneapolis, Minnesota, United States: ACM, 2000.

[48] S. J. Mellor, A. N. Clark, and T. Futagami, "Guest Editors' Introduction: Model-Driven Development," *IEEE Softw.,* vol. 20, pp. 14-18, 2003.

[49] X.-S. Yang and S. Deb, "Cuckoo search via L´evy flights," in *Proceedings of World Congress on Nature & Biologically Inspired Computing (NaBIC 2009)*, 2009, pp. 210-214.

[50] O. Räihä, "A survey on search-based software design," *Computer Science Review,* vol. 4, pp. 203 - 249, 2010.

[51] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed., 2002.

[52] J. Clarke, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd, "Reformulating software engineering as a search problem," *Software, IEE Proceedings,* vol. 150, pp. 161-175, 2003.

[53] S. M. Sait and H. Youssef, *Iterative Computer Algorithms with Applications in Engineering: Solving Combinatorial Optimization Problems*: Wiley-IEEE Computer Society Press, 2000.

[54] S. Santini and R. Jain, "Similarity measures," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 21, pp. 871-883, 1999.

[55] Y. Wang and N. Ishii, "A Method of Similarity Metrics for Structured Representations," *Expert Systems with Applications,* vol. 12, pp. 89-100, 1997.

[56] J. Munkres, "Algorithms for the Assignment and Transportation Problems," *Journal of the Society for Industrial and Applied Mathematics,* vol. 5, pp. 32-38, 1957.

[57] Y. Wang and N. Ishii, "A genetic algorithm and its parallelization for graph matching with similarity measures," *Artificial Life and Robotics,* vol. 2, pp. 68-73, 1998.

[58] A. H. El-Maleh, S. M. Sait, and A. Bala, "Cuckoo Search Optimization in State Assignment for Area Minimization of Sequential Circuits," in *29th International Conference on Computers and Their Applications CATA-2014*, Las Vegas, USA, 2014.

[59] N. Nakatsu, Y. Kambayashi, and S. Yajima, "A Longest Common Subsequence Algorithm Suitable for Similar Text Strings," *Acta Informatica,* vol. 18, pp. 171 - 179, 1982.

[60] A. Begum, "A Greedy Approach for Computing Longest Common Subsequences," *Journal of Prime Research in Mathematics,* vol. 4, pp. 165-170, 2008.

[61] M. H. Alsuwaiyel, *Algorithms: Design Techniques and Analysis*: World Scientific Publishers, 1999.

[62] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave, "Matching and Merging of Statecharts Specifications," in *29th International Conference on Software Engineering (ICSE 2007)*, 2007, pp. 54-64.

[63] H. O. Salami and M. A. Ahmed, "A framework for reuse of multi-view UML artifacts," *International Journal of Soft Computing and Software Engineering,* vol. 3, pp. 156 - 162, 2013.

[64] M. Genero and M. Piattini, "Empirical validation of measures for class diagram structural complexity through controlled experiments," in *5th International ECOOP Workshop on Quantitative Approaches in object-oriented Software Engineering (QAOOSE 2001)*, 2001.

[65] J. Muskens, M. Chaudron, and C. Lange, "Investigations in applying metrics to multi-view architecture models," in *30th Euromicro Conference*, 2004, pp. 372 - 379.

[66] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.,* vol. 2, pp. 308 - 320, 1976.

[67] M. Genero, D. Miranda, and M. Piattini, "Defining and validating metrics for UML statechart diagrams," *Data and Knowledge Engineering,* vol. 64, pp. 534-557, 2008.

[68] A. Mili, R. Mili, and R. T. Mittermeir, "A survey of software reuse libraries," *Ann. Softw. Eng.,* vol. 5, pp. 349-414, 1998.

[69] S. Teufel, "An overview of evaluation methods in TREC ad hoc information retrieval and TREC question answering," *Evaluation of Text and Speech Systems,* pp. 163-186, 2007.

[70] V. Basili, L. C. Briand, S. Condon, Y.-m. Kim, W. L. Melo, and J. D. Valettt, "Understanding and Predicting the Process of Software Maintenance Releases," in *in proceedings of the 18th international conference on software engineering*, 1996, pp. 464-474.

[71] D. Zhang and J. J. P. Tsai, *Advances in Machine Learning Applications in Software Engineering*: IGI Global, 2007.

[72] T. Yue, L. Briand, and Y. Labiche, "Automatically Deriving UML Sequence Diagrams from Use Cases," Simula Research Laboratory 2010.

[73] L. A. Zager and G. C. Verghese, "Graph similarity scoring and matching," *Applied Mathematics Letters,* vol. 21, pp. 86 - 94, 2008.

[74] V. Bandaru and S. D. Bhavani, "Graph Isomorphism Detection Using Vertex Similarity Measure," in *Communications in Computer and Information Science*. vol. 168, S. Aluru, S. Bandyopadhyay, U. Catalyurek, D. Dubhashi, P. Jones, M. Parashar, and B. Schmidt, Eds.: Springer Berlin Heidelberg, 2011.

[75] H. K. Wright, M. Kim, and D. E. Perry, "Validity concerns in software engineering research," in *FSE/SDP workshop on Future of software engineering research*, Santa Fe, New Mexico, USA, 2010, pp. 411-414.

[76] M. d. O. Barros and A. C. Dias-Neto, "Threats to Validity in Search-based Software Engineering Empirical Studies," *RelaTe-DIA,* vol. 5, 2011.

Vitae

Hamza Onoruoiza Salami was born on October 7, 1980 in Jos, Plateau State, Nigeria. He completed his B. Tech and MSc in Computer Science from Abubakar Tafawa Balewa University, Bauchi, Nigeria in 2002 and 2010, respectively. Mr. Hamza has worked as a lecturer in Mathematical Sciences Department of Nasarawa State University, Keffi, Nigeria from 2004 to 2010. He joined King Fahd University of Petroleum and Minerals (KFUPM), Dhahran, Saudi Arabia in 2010. He just completed his PhD in Computer Science and Engineering (CSE) at KFUPM in 2014.

Mr. Hamza has presented papers at two international conferences in the United States:

- The International Conference on Soft Computing and Software Engineering 2013 (SCSE'13), San Francisco, California, USA, March 1 – 2, 2013,
- The 12th International Conference on Machine Learning and Applications (ICMLA 2013), Miami, Florida, USA, Dec 4 – 7, 2013.

Hamza Salami is a Nigerian citizen and can be contacted at:

Permanent Address: No. 10, Bello Sidi-Ali Street, Madalla, Niger State, Nigeria.

Email: *hamzasalami@yahoo.co.uk*