

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

**ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600**

UMI[®]



NEW STRUCTURAL SIMILARITY METRICS FOR UML MODELS

BY

RAIMI AYINDE RUFAI

A Thesis 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

MASTER OF SCIENCE

In

COMPUTER SCIENCE
JANUARY 2003

UMI Number: 1413039

UMI[®]

UMI Microform 1413039

Copyright 2003 by ProQuest Information and Learning Company.

**All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.**


**ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346**

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES


This thesis, written by **RAIMI AYINDE RUFAl** under the direction of his thesis advisor and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER SCIENCE**.


Thesis Committee


Dr. Moataz Ahmed (Chairman)


Dr. Muhammad Alsuwaiyel (Member)


Dr. Jarallah Alghamdi (Member)


Department Chairman


Dean of Graduate Studies

19-3-2003

Date



Thesis Abstract

NAME: Raimi Ayinde Rufai

**TITLE: NEW STRUCTURAL SIMILARITY METRICS FOR UML
MODELS**

MAJOR FIELD: COMPUTER SCIENCE

DATE OF DEGREE: MARCH 2003

This thesis proposes a conceptual framework for composing a multi-view similarity metric from a set of similarity metrics, each focusing on a particular view of software. Software has at least three views: structural, behavioral, and use-case views. This thesis also proposes a set of structural similarity metrics for Unified Modeling Language (UML) models. The design and implementation of a proof-of-concept UML model comparison tool based on these structural similarity metrics are reported. Results from conducted case studies demonstrate the potential of the proposed metric set.

خلاصة الرسالة

الاسم : رحيمي أيندارفاعي.

عنوان الدراسة : مقاييس جديدة للتشابه البنائي لنموذج UML.

النخصص : علوم الحاسب الآلي.

تاريخ التخرج : يناير 2003.

يقترح هذا البحث إطاراً تصورياً لتكوين مقياس تشابه متعدد الرؤى من مجموعة مختلفة من مقاييس التشابه، كلٌّ يركز على رؤية محددة من برامج الحاسب الآلي. يحوي البرنامج على الأقل ثلاثة رؤى: رؤى بنائية، رؤى سلوكية، ورؤى حالة الإستعمال. كما يقترح هذا البحث مجموعة من مقاييس التشابه البنائية لنماذج UML. تم إعداد تقرير يحوي تصميم وتطبيق أداة البرهان التصوري لمقارنة نماذج UML والمبني على هذه المقاييس التشابه البنائية. إن النتائج المستخلصة لعدد من الدراسات تظهر إمكانية استعمال هذه المقاييس المقترحة.

درجة الماجستير في العلوم

جامعة الملك فهد للبترول والمعادن

يناير 2003

Table of Contents

Thesis Abstract.....	iii
خلاصة الرسالة.....	iv
Table of Contents	v
List of Figures	vii
List of Tables.....	viii
Acknowledgments.....	ix
Introduction.....	1
1.1 Software Reuse.....	1
1.1.1 Reusable Software Artifacts.....	3
1.1.2 Software Reuse Processes	6
1.1.3 Software Retrieval.....	7
1.2 Software Product Lines	9
1.3 Motivation	10
1.4 Main Contributions	11
1.5 Organization of Thesis	12
Literature Survey.....	13
2.1 Introduction	13
2.2 Related Work	13
2.2.1 Metric-Based Techniques.....	14
2.2.2 Deduction-Based Techniques.....	24
2.2.3 Classification-Based Techniques	26
2.3 The Unified Modeling Language (UML).....	29
Conceptual Framework	35
3.1 Introduction	35
3.2 Multiple Views of Software	36
3.3 Fundamental Assumptions	37
3.4 Metric Composition and Cascading	39
3.5 Computing the Inconsistency Penalty	41

Structural Similarity Metrics	43
4.1 Introduction	43
4.2 Algorithms for Class Model.....	43
4.2.1 Semantic Distance Measure	44
4.2.2 Shallow Semantic Similarity Metric	47
4.2.3 Deep Semantic Similarity Metric	51
4.2.4 Signature-Based Similarity Metric	54
4.2.5 Relationships-Based Similarity Metric	58
4.3 Implementation	60
Evaluation	65
5.1 Introduction	65
5.2 Theoretical Validation.....	65
5.2.1 Theoretical Validation of SSSM	66
5.2.2 Theoretical Validation of DSSM.....	67
5.2.3 Theoretical Validation of SBSM.....	69
5.2.4 Theoretical Validation of RBSM	70
5.3 Empirical Validation	71
5.3.1 Our Intuition about Similarity	72
5.3.2 Goals, Hypothesis and Theories	73
5.3.3 Experiment Plan	74
5.3.4 Results	79
Conclusion.....	112
6.1 Introduction	112
6.2 Summary and Contributions of Thesis.....	112
6.3 Limitations and Further Work.....	113
Bibliography.....	116
Index.....	124
Appendix A : Tool Design	127

List of Figures

<i>Number</i>	<i>Page</i>
Figure 1: A Software Retrieval System.....	8
Figure 2: Tree Structure of Diagram Categories	39
Figure 3: Metric Cascading	40
Figure 4: The Filter Metaphor	62
Figure 5: The Composite Filter Pattern Applied to Class Similarity Metrics	64
Figure 6: Comparing JDK versions using SSSM.....	81
Figure 7: Comparing ANT versions using SSSM.....	87
Figure 8: Comparing JDK versions using SBSM	90
Figure 9: Comparing ANT versions using SBSM.....	97
Figure 10: Comparing JDK versions using RBSM.....	100
Figure 11: Comparing ANT versions using RBSM	107

List of Tables

Table 1: Summary of Metric-Based Software Component Retrieval Techniques ...	23
Table 2: Four-layer Metamodeling Architecture [77].....	31
Table 3: Table of features for different WordNet-based similarity measures.....	45
Table 4: Experimental Design for H_3	76
Table 5: Comparing JDK and J2SDK Versions using SSSM.....	80
Table 6: Correlation between proportionate changes in number of classes and SSSM values in the JDK versions.....	84
Table 7: Correlation between proportionate changes in number of classes and SSSM values in the ANT versions.....	85
Table 8: Comparing Apache ANT Versions using SSSM.....	86
Table 9: Rank Correlation Computation for SSSM.....	88
Table 10: Comparing JDK and J2SDK Versions using SBSM.....	89
Table 11: Correlation between proportionate changes in # of classes and SBSM values in the JDK versions.....	93
Table 12: Correlation between proportionate changes in # of classes and SBSM values in the ANT versions.....	94
Table 13: Comparing Apache ANT Versions using SBSM.....	96
Table 14: Rank Correlation Computation for SBSM.....	98
Table 15: Comparing JDK and J2SDK Versions using RBSM.....	99
Table 16: Correlation between proportionate changes in number of classes and RBSM values in the JDK versions.....	103
Table 17: Correlation between proportionate changes in number of classes and RBSM values in the ANT versions.....	104
Table 18: Comparing Apache ANT Versions using RBSM.....	106
Table 19: Rank Correlation Computation for RBSM.....	108
Table 20: Comparing JDK against Apache ANT.....	109
Table 21: Distribution of Naming Anomalies.....	111

Acknowledgments

All thanks are due Allah first and foremost for his countless blessings. Acknowledgement is due to King Fahd University of Petroleum & Minerals for supporting this research.

My unrestrained appreciation goes to my advisor, Dr. Moataz Ahmed, for all the help and support he has given me throughout the course of this work and on several other occasions. I also wish to thank my thesis committee members, Dr. Muhammad H. Alsuwaiyel and Dr. Jarallah AlGhamdi, for their help, support, and contributions. I simply cannot begin to imagine how things would have proceeded without their help and support.

I would also like to thank Prof. David Rine (George Mason University), Prof. Ernesto Damiani (University of Milan), Dr. Jacob Cybulski (Deakin University), and Dr. Philip Bernstein (Microsoft) for valuable discussions.

I also acknowledge my many colleagues and friends who have helped with this work in some way. Especially noteworthy is Sohail Khan, for valuable discussions.

Finally, I wish to express my gratitude to my family members for being patient with me and offering words of encouragements to spur my spirit at moments of depression.

Chapter 1

Introduction

1.1 Software Reuse

One of the earliest works on software reuse is the seminal paper of McIlroy [61], which was an invited talk at the 1968 NATO Software Engineering Conference. In that talk, McIlroy made two important observations. First, he observed that the software industry was far from being industrialized, in the sense that mass production techniques for software were lacking. Second, he noted that there was no notion of a software component sub-industry, i.e. software houses that develop reusable software components. These early ideas have fueled the interests of many later researchers. Later works [100][51][60][9][67] have developed these ideas into what is now known as the field of software reuse.

Different authors have coined their own definitions of the term, software reuse. The following are some notable ones:

- *Software Reuse* [66] is the *process* whereby an institution defines a set of *systematic* operating procedures to produce, classify and retrieve *software artifacts* for the purpose of using them in its development activities.

- *Software Reuse* [51] is the *process* of creating software systems from *existing software* rather than building software systems from scratch.

A commonality of these definitions is that software reuse is a *process* and that the process has *software artifacts* as one of its inputs. What is not obvious from the definitions is that software reuse is also a field, a discipline [4] and a paradigm of software development [96]. These extensions are well known, for instance, Sommerville [96] defines software reuse as “an approach to software development which tries to maximize the reuse of existing software.” However, these extensions are not of interest to us in this work. Of interest to us are the software reuse process and the development artifacts that form the input to that process. Before proceeding further, we would like to address the fundamental question: *Why should we reuse software?*

Software reuse has clear benefits, although it does suffer from a few problems too. Apart from reduction in overall development costs, other reuse benefits include increased reliability, reduced process risk, effective use of specialists, standards compliance, and accelerated development [96]. Reuse problems include increased maintenance costs, the not-invented-here syndrome, *lack of tool support*, *difficulty of maintaining a library of reusable artifacts*, and *the cost of locating and adapting reusable artifacts* [96].

A step towards a solution to the last three problems, italicized above, is an effective software retrieval system. Retrieval is one of the activities in a *software reuse process*, which takes in a *query* as input and returns *reusable artifacts* (or objects of reuse) as output. Because the goal of software retrieval is to return reusable software artifacts, we first introduce reusable software artifacts in the sequel, before going on to introduce reuse processes.

1.1.1 Reusable Software Artifacts

It is customary in the software reuse literature to make the distinction between the *generative or reusable processor* approach and the *building blocks* approach. In the generative (or reusable processor) approach, developers reuse development processors such as code generators, wizards, or high-level specification language interpreters. In the building blocks approach, the objects of reuse are the products of previous software development efforts (called software artifacts) such as source code, design, requirements specification, and analysis models. In this work, our focus is on the building block approach. In the building blocks approach, different types of reusable artifacts may form the input to the reuse process.

The following are classes of reusable artifacts:

- *Domain Models*: These can be reused at the earliest stage of the software development process, the domain analysis stage. Very few

systems exist that exploits the reuse of artifacts at this stage. An example of such a system is the work of Blok and Cybulski [10]. Another is the generic application frames in the ITHACA development environment [24]. Yet, a more recent example can be found in the software product lines approach, which is often touted as the newest silver bullet for actualizing software reuse goals [87][12][17][38].

- *Requirement Specifications:* These artifacts can be reused during the requirements analysis phase of the software lifecycle [19]. An example of how a requirements specification reuse may be assisted by a software tool is described in a recent paper by Cybulski and Reed's [18].
- *Design:* These artifacts can be reused during the design phase. An example of a design repository is the SPOOL Design Repository [45]. Another is the work of Lee and Harandi [55].
- *Documentation:* All documentations, both technical and non-technical documentations about a development project and the product can be reused [87][12].
- *Test Data:* The test data generated for one project can be reused in another project if they are sufficiently similar [87].

- *Code*: Source is perhaps the most often reused part of a software product [19]. Examples of code reuse abound around us. Works on automated support for code reuse include works of Michail and Notkin [64] and Sarireta and Vaucher [91].

We refer to the first three items on the above list as *early-stage reusable artifacts* and refer to the rest as *later-stage reusable artifacts*. Early-stage reusable artifacts are particularly beneficial, because once a match is found for them, related later-stage artifacts for the match can also be reused. For instance, if an analysis model for a previous project A contained in a project repository is found to match the analysis model of the current project (the query), then its design, code and documentation may be reused in the current project.

In this work, our focus is on the early-stage reusable artifacts of the software development lifecycle that are represented using the Unified Modeling Language (UML). The UML is a language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system [11]. We refer to the UML representations of these artifacts as *UML models*. In our work, UML models form an input to the reuse processes, which we discuss in the sequel.

1.1.2 Software Reuse Processes

In the context of software reuse, there are three processes to consider:

- *Developing for reuse* [67][10]: This refers to the process of developing reusable artifacts.
- *Developing with reuse* [67][10]: This refers to the process of developing with reusable artifacts. It involves locating reusable artifacts, possibly modifying them and then integrating them into the current system.
- *Reuse within development* [104][25]: This marks a paradigm shift from the preceding process. In this paradigm, reuse is inherently part of the development process. Task-relevant components from the repository are automatically displayed in the programming environment.

Because details of the first are outside the scope of this thesis, we invite interested readers to consult references [67][10]. While our work may be directly relevant to the remaining two, we shall assume in the rest of this work that our target is “developing with reuse.” Developing with reuse consists of the following activities [67]: locating reusable artifacts (retrieval), assessing their relevance to current needs (assessment), and adapting them to those needs (adaptation). Locating reusable artifacts often involves some

form of comparison of a query with candidate models in the repository. Assessment and adaptation shall not be discussed further, since these are outside the scope of our work. In the sequel, we discuss software retrieval.

1.1.3 Software Retrieval

Because the volume of reusable artifact retrieval systems available in the literature is daunting, frameworks and taxonomies are needed to understand the works and put them in proper perspective. In the next chapter, we shall present a survey of related work. Meanwhile, we present an abstract model of a software retrieval system.

In order to reuse a software artifact, we first have to locate it. Locating artifacts is essentially a search problem. In every search, a *search space*, a *search goal*, and a *comparison function* are always defined. In software retrieval, the search space is known as the *software repository*. The search goal is called the *query*. The comparison function is called a *similarity metric*. Repositories often house a large collection of artifacts or *representations* of them (called *surrogates* [65]). How well the retrieved artifacts match the query depends on the soundness of the *similarity metric*.

In this work, we, first, investigate existing similarity measures between query artifacts (i.e., the artifacts described by the query) and artifacts in the repository and then propose a structural similarity metric for UML models,

while laying the foundations for a multi-view similarity metric (i.e. a similarity metric that considers multiple views of software in its similarity assessment).

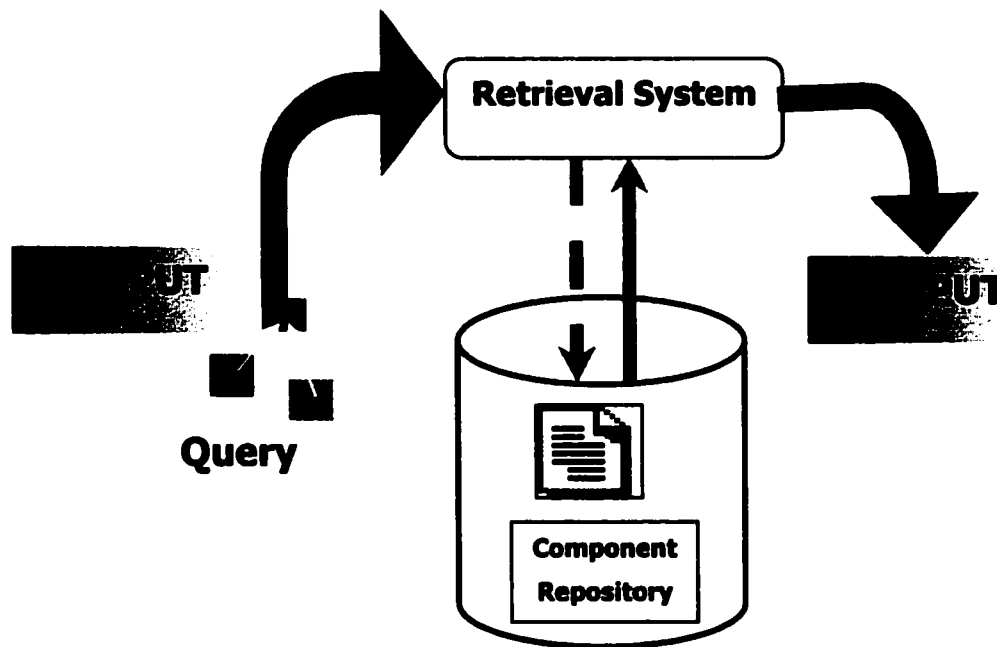


Figure 1: A Software Retrieval System

It is noteworthy to mention that the reuse process as described thus far can be likened to a traditional library system. The repository is the library building. The artifacts correspond to the monographs, and other library materials stored in it. Retrieval may be likened to borrowing. Because of this similarity, some authors have described these approaches to practicing software reuse as library-based reuse approaches (e.g. see Martin Griss' paper [39] and Rine's [87]). These authors have often criticized the so-called

library metaphor of reuse and consider it as a mistake [39][87]. A relatively newer metaphor for software reuse is now regarded as the magical wand that will actualize all the goals of software reuse. The metaphor is that of an assembly line in a manufacturing factory. This factory metaphor of software development is more commonly known as the software product lines (SPL) approach, which the next section briefly introduces.

1.2 Software Product Lines

In this section, we examine the software products lines approach, which has more or less become the consensus of today's software engineering community as the most effective way to practice software reuse [12][87].

Over the last decade, there has been a paradigm shift from library-based reuse programs, such as what we have presented so far, towards the software product lines (SPL) approach [39][87]. A software product line is a set of software-intensive systems that share a common, managed feature set satisfying a particular market segment's specific needs or mission and that are developed from a common set of core assets in a predefined way [17].

Organizations practicing SPL have often identified two development roles: domain engineering and application engineering [72]. Domain engineering is the development and maintenance of the shared assets across a product line,

while application engineering is the development of products in the product line using the shared assets. The former is analogous to development for reuse, while the later is to development with reuse (*cf.* 1.1.2).

Organizations needing to migrate to a product line approach, will often have to carry over their set of pre-existing assets, a process known in the SPL literature as *asset mining* [27][8][7]. Asset mining usually occurs after an organization divides its products into groups based on the size of the common feature sets. This is with the aim of defining the product families of an organization (scoping [92]).

Scoping also occurs at the asset level (asset-centric scoping [92]) which means deciding what assets should be part of the shared product line infrastructure. Some of these assets may have to be developed from scratch or purchased from a vendor or *mined* from a set of legacy assets. Asset mining [27][8][7] involves finding out what assets should be carried over from among the preexisting assets into the shared asset base for a particular product family.

1.3 Motivation

As noted earlier, the problems with reuse include lack of tool support, difficulty of maintaining a library of reusable artifacts, and the cost of

locating and adapting reusable artifacts. These have long been recognized as the fundamental problems with reuse [61][51][9].

Even though the UML has become more or less the *de facto* standard modeling language for representing analysis as well as design artifacts [49], researchers have done little in proposing a similarity metric for UML models [88].

Such a metric would have clear benefits. Besides forming the comparison function for a retrieval system, a similarity metric can also serve the need of a designer who wants to verify the code developed by his programmers against the design (tracing) [3][96]. The steps involved in the case of tracing would be to first reverse-engineer the source code into a UML model. Our metric can then be used to match the reengineered model to the design, in order to detect possible inconsistencies. Further, the metric will help in the clustering of products that will be migrated to the same product line. The metrics can also be used in asset mining. These clear benefits have fueled our motivation to investigate this problem.

1.4 Main Contributions

The main contributions of this thesis work are the following:

- Conducting an extensive critical survey.

- Proposing a conceptual framework for composing multi-view similarity metric from a set of similarity metrics, each of which focuses on a particular view of software.
- Proposing a set of structural similarity metrics for UML models.
- Designing and building a proof-of-concept UML model comparison tool based on these metrics.
- Conducting case studies to demonstrate the soundness (or lack thereof) of a subset of the proposed metrics.

These findings have been reported in this thesis.

1.5 Organization of Thesis

The rest of this thesis is organized as follows. Chapter 2 gives an extensive literature survey of related work. Chapter 3 presents the conceptual approach that we have taken. Chapter 4 presents implementation details of the metrics we have developed. Chapter 5 gives the result of experiments. Chapter 6 gives the conclusion and further work.

Chapter 2

Literature Survey

2.1 Introduction

In this chapter, we discuss related work and introduce the Unified Modeling Language (UML), along with its cross-product models interchange problem .

2.2 Related Work

In this section, we present related work we have found in the literature. We discuss each work with respect to the following attributes: *artifact representation*, *query representation*, and *matching protocol*.

Artifact representation refers to the abstraction that represents the artifact in the repository. Query representation refers to the structure of the query sent to the retrieval system. Sometimes, both representations are identical, but not always. By matching protocol, we mean how queries are matched to artifacts.

Since the target of our work is the matching protocol, we use it to classify previous works into the following classes: *metric-based techniques*, *deduction-based techniques*, and *classification-based techniques*. Metric-based techniques refer to matching artifacts based on a measure of similarity.

Deduction-based techniques rely on deductive reasoning to determine matches. Classification-based techniques partition artifacts into cluster hierarchies based on domain ontology, allowing matching artifacts to be found by navigation through a unique path in the hierarchy. The metrics that we shall be proposing in this thesis belong to the first category.

In the following sections, we discuss related works using this classification.

2.2.1 Metric-Based Techniques

Some of the earliest works on metric-based matching of artifacts were for matching source codes. Ottenstein [79], in 1976, published his work on comparing FORTRAN programs for similarity. He used the four basic Software Science measures proposed by Halstead [40] as a measure of program similarity. Grier [37] reports an approach, in which he uses twenty different measures (Halstead's inclusive) for judging Pascal program similarity. Other works of this nature that appeared later include those of Aiken [1], Whale [101], Verco and Wise [99], and Prechelt *et al.* [83]. In all these works the artifact representation as well as the query representation is the source code. All these works are useful, for the purpose of which they have been designed: detecting plagiarized source codes. However, this is not very useful to the reuse community as it might sometimes be more cost-effective to build source code from scratch than having to go through the

time consuming activities of code understanding and retrofitting. In the next paragraph, we give some details of the Prechelt *et al.* approach since it is the latest and in a sense represents the state of the art in source code matching.

The work of Prechelt *et al.* [83] describes a tool (called JPlag) for detecting plagiarism among programming assignment submissions. JPlag starts out by converting the programs to be compared into an intermediate representation (token strings). These token strings are compared in pairs for determining the similarity of each pair. The lengths of the maximum matchings discovered are used to compute the degree of similarity using the equation below (Equation 1). The worst-case complexity of the comparison algorithm is $O(n^3)$.

$$sim(A, B) = \frac{2 \cdot \sum length}{|A| + |B|} \quad 1$$

where A and B are programs to be compared for similarity. This algorithm has good computational complexity. However, it is aimed at comparing source code to source code. Given the goal of the authors is to detect program plagiarisms, their approach is barely suitable for comparing models that exist at a higher level of abstraction.

Lee and Harandi [55] proposed an approach to retrieve domain models and design components from a reuse repository. Their representation of design models and components is a two-part representation. For convenience, we

refer to both as designs. Lee and Harandi represent each design as a DFD and an ER diagram, where the DFDs represent the process component and the ER diagrams represent the data component. Lee and Harandi define a mapping from this representation to the object model by considering each data store in the DFD and entity types in the ER diagrams as an object type. The attributes from the ER diagram form the attributes in the object model. Transformations from the DFD diagrams represent operations in the object model. Further, Lee and Harandi store these designs in a knowledge base, which also contains some background knowledge. The background knowledge includes a basic object lattice, a datatype lattice (i.e. an *isA* hierarchy), and is-part-of hierarchies for composite types. Lee and Harandi define a distance metric [56] as follows:

$$SIM = \left(\alpha \sum (MaxScore_{object_type} - ObjectScore) + \beta \sum (MaxScore_{attribute} - AttributeScore) + 1 \right)^{-1} \quad 2$$

where α and β are weights attached to object type similarity and attribute-based similarity respectively; *ObjectScore* is a score to indicate the degree of match between the two object types; *AttributeScore* is a score indicating the degree of match between the attributes of the two object types. *MaxScore_{object_type}* is computed as the maximum of the following two: length of the longest *isA* path in the object type lattice and the maximum number of attributes expected in an object type. The *MaxScore_{attribute}* is computed as

l + length of the longest path in the data type lattice. It is interesting to point out here that Lee and Harandi have not considered operations in their distance assessment.

Girardi and Ibrahim [35] developed a tool called ROSA (Reuse Of Software Artifacts) for automatic indexing and retrieval of reusable software artifacts. Users of the tool, both indexers and retrievers give natural language descriptions of the reusable artifacts. These are then converted into frame-like representations. Retrieval is achieved using a similarity measure (shown in Equation 3) that compares the frame representations of queries with those of components stored in the repository. Every component c in the repository is represented by a set of natural language descriptions. Each such description d , has a number of interpretations, l .

$$S(q, c) = \max_{kdl} \left(\sum_{\forall j \in SC_{qkj} \cap SC_{cdlj}} w_j \cdot sc_closeness(SC_{qkj}, SC_{cdlj}) \right) \quad 3$$

A query, q , is usually given in a single description, k , which could have a set of interpretations, j , too. Equation 3 above shows their metric between query q and component c . w_j represents the weight attached to the interpretation j . The function *sc_closeness* computes the similarity between two interpretations.

The work of Damiani *et al.* [22] does capture some semantic information and takes care of synonyms by incorporating a thesaurus into their matching model. Their representation for a component is a *software descriptor* [22]. Their technique is interesting because it does recognize the fact that the process of component retrieval is fraught with uncertainty. Conceptually, the notion of applying fuzzy reasoning techniques [48] to handle the inherent uncertainty in feature terms is appealing. However, Damiani *et al.* have used singleton fuzzy sets. This way, their work more or less reduces to using crisp values for fuzzy linguistic variables. Although good empirical results are currently being obtained from their system, we are optimistic that when the system is modified to use non-singleton fuzzy membership functions, even better results will be possible.

Sarirete and Vaucher [91] proposed a similarity measure based on the semantic relation between the words that identify the concepts in the object model. WordNet, a widely available computer thesaurus, has been used to guess the semantic relations between terms. There are three problems with this approach. First, it concentrates only on the static view of a software artifact (the class model) and ignores all other views. Second, the approach as described in their paper is computationally intensive. Third, the inherent ambiguity of natural language and the chance that developers and analysts

may not always choose descriptive names can limit effectiveness of their approach.

$$Sim(N_1, N_2) = \begin{cases} 1 & N_1 \text{ and } N_2 \text{ are identical} \\ \frac{1}{2^n} & N_1 \text{ and } N_2 \text{ are hypernyms; } n = \text{level between concepts} \\ 1 - \frac{1}{2^d} & N_1 \text{ and } N_2 \text{ are synonyms; } d = \text{number of common concepts} \\ -1 & N_1 \text{ and } N_2 \text{ are antonyms} \end{cases} \quad 4$$

The work of Antoniol *et al.* [3] has the goal of improving code traceability by extracting design information from source code and then comparing it with the original design. The output is a measure of similarity associated to each matched class, plus a set of unmatched classes. Both the initial design and that extracted from the source code are represented in a custom OO design description language called AOL (Abstract Object Language). Matching of design classes with code classes reduces to textual matching of attribute and method names using string edit distance. Matching class names, attribute names and method names may be perfect for the goal of tracing design into code, but can hardly serve the purpose of matching models in a sound way.

Michail and Notkin [64], borrowing heavily from the field of Information Retrieval (IR), represents components (which are mainly modules and functions) as a vector of weighted terms. Thus, a component D_i will be represented as the vector of pairs, $T_i = \{(t_{i,1}, w_{i,1}), (t_{i,2}, w_{i,2}), (t_{i,3}, w_{i,3}), \dots\}$

where t_{ij} is the j -th term describing the component D_i and w_{ij} is its weight. This representation is identical with that used in the SMART Retrieval System [89]. They compute a measure of similarity using an unnormalized variant (equation 5) of the cosine similarity measure (equation 6) of Salton and McGill [89].

$$S_{i,j} = \sum_t w_i(t).w_j(t) \quad 5$$

$$S_{i,j} = \frac{\sum_t w_i(t).w_j(t)}{\sqrt{\sum_t w_i^2(t). \sum_t w_j^2(t)}} \quad 6$$

The problem of obtaining terms and generating weights using frequency analysis techniques make this representation nontrivial [13]. One disadvantage of this similarity measure is that it requires that terms match exactly. Consideration is not given to synonymous term. One improvement to the representation is the various kinds of faceted and controlled vocabulary schemes and the incorporation of some kind of thesaurus.

Blok and Cybulski [10] proposed a method of matching UML models by means of their sequence diagrams. Event vectors are used to represent sequence diagrams. In other words, each UML model is represented by the set of event vectors that represent its sequence diagrams. Although, an event vector could simply be represented by a sequence of methods calls made in a particular sequence diagram, the authors have not chosen this option because

they believe computing semantic distance between the events would be computationally expensive. Instead, they have chosen to obtain it by a series of steps. First, all events in the domain are assigned to different clusters based on their semantics. Events from the query model are also assigned to clusters. Numeric identifiers are used to name clusters. An event vector is the vector of clusters to which its events belong. The event vectors in the query model is extracted and matched with *likely candidates* from the repository using the metric shown below. Because determining likely candidates can be hard, the authors have employed the following heuristics. First, they sort all the event flows of a use case in increasing order of their length, i.e. in terms of the number of events. Then, they take the first use case and then find a match for it. This heuristic will prevent having to compare every use-case in the query model to every other ones in the target model. This heuristic however does not always produce the optimal matchings [20].

$$similarity(U, V) = \frac{\sum_{i=1}^x similarity(Q_u, D_{vt}, I)}{x} \quad 7$$

$$similarity(Q, D, I) = \frac{\sum_{c=1}^n q_c \cdot d_c \cdot i_c}{\sqrt{\sum_{c=1}^n (i_c \cdot d_c^2) \cdot \sum_{c=1}^n (i_c \cdot q_c^2)}} \quad 8$$

The first equation defines the similarity metric for matching the query model

U and a model V from the repository. Since there will be several use-cases within each model. The number of use-cases matched depends on the size of the smaller model (as measured by the number of use-cases), i.e. $x = \min(|U|, |V|)$. Q_{ut} and D_{vt} denote the event vector t from query use case, U and the event vector t from the use-case, V , from the domain model respectively. I is the importance vector, computed from the frequency of occurrence of a cluster in a domain model. The second equation defines how the similarity between events vectors is computed.

The following table summarizes the works presented here and states their limitations.

Table 1: Summary of Metric-Based Software Component Retrieval Techniques

Work	Artifact Type	Query Representation	Metric	Limitation
Prechelt <i>et al.</i> , 2002 [83]	Source Code	Source Code	$sim(A, B) = \frac{2 \cdot \sum length}{ A + B }$	Applies only to source code
Lee and Harandi, 1993 [55]	DFD and ER Diagrams of domain, analysis and design models	DFD and ER query design model	$SIM = \left(\alpha \sum (MaxScore_{object_type} - ObjectScore) + \beta \sum (MaxScore_{attribute} - AttributeScore) + 1 \right)^{-1}$	Not UML
Girardi and Ibrahim, 1995 [35]	NL Representations of domain, design, and analysis models,	Natural language queries	$S(q, c) = \max_{kdl} \left(\sum_{v \in SC_q \cap SC_{c,q}} w_j \cdot sc_closeness(SC_{qj}, SC_{c,qj}) \right)$	Assumes the indexer will give good NL descriptions to models
Sarireta and Vaucher, 1997 [91]	Class models	Source code	$Sim(N_1, N_2) = \begin{cases} 1 & N_1 \text{ and } N_2 \text{ are identical} \\ \chi_c & N_1 \text{ and } N_2 \text{ are hypernyms; } n = \text{level between concepts} \\ 1 - \chi_c & N_1 \text{ and } N_2 \text{ are synonyms; } d = \text{number of common concepts} \\ -1 & N_1 \text{ and } N_2 \text{ are antonyms} \end{cases}$	Only class models is considered
Michail and Notkin, 1999 [64]	Modules and functions	Vector of weighed terms	$S_{i,j} = \sum_i w_i(i) \cdot w_j(i)$	Mere string matching; not UML
Blok and Cybulski, 1998 [10]	Use cases and sequence diagrams of domain and analysis models are represented as event vectors. Could also work for design models	Same as artifact	$similarity(Q, D, I) = \frac{\sum_{i=1}^n q_i \cdot d_i \cdot i_i}{\sqrt{\sum_{i=1}^n (i_i \cdot d_i^2) \cdot \sum_{i=1}^n (i_i \cdot q_i^2)}}$	UML, but class models are not considered.

2.2.2 Deduction-Based Techniques

Perhaps, one of the earliest works on deduction-based retrieval is Rich and Waters' work on the Programmer's Apprentice project [86]. They have represented high-level programming abstractions (or what they termed *clichés*) using a formalism called the Plan Calculus. The Plan Calculus is actually layered on top of frame-like representation. Retrieval is achieved using a combination of general-purpose predicate logic reasoning and special-purpose algebraic deduction. The target of the work is locating program code.

Zaremski and Wing [105] described signature matching in the context of software libraries. They defined a several flavors of function and module matching. These matchings are defined formally using match predicates. Retrieval reduces to reasoning through the software library to find matches for the query based on the match predicate chosen.

Mili *et al.* [69], also working in the context of software libraries, represented software by its formal functional specification and defined an ordering relation between specifications called the refinement ordering, which as one would expect, is a partial order relation. The partial order relations are said to have lattice properties, which means that only the subset of the specifications that minimizes the *meet* and maximizes the *join* between the query

specification and the specifications, need to be examined during retrieval. By varying the definition of the *meet* relation, a number of approximate matching predicates were defined. The target of the work was software program libraries.

Schuman and Fischer [94] described a tool called NORA/HAMMR, which uses a pipeline of deductive filters to locate matches from a component repository, given a query. Both the query and the components resident in the repository are represented in VDM-SL, which is a specification language. Thus, users will have to be able to specify their queries in this language. The tool provides a default set of predicates for matching while allowing users to specify new predicates (or matching relations) as well. Predicates (called filters) are applied one after another, with the output of one filter feeding into the next in a pipeline fashion. This allows for anytime behavior, whereby matches can be produced whenever requested even before the whole pipeline has been traversed.

Jilani et al. [43] made a distinction between functional similarity and structural similarity. The authors argued that while there are situations where structural similarity measures are desirable (e.g. during retrieval for adaptation), real world constraints limit their practicality. Thus, they went on to discuss measures of distance that depend on functional properties of

software artifacts. Their representation is the functional specification of the artifact. While functional specifications often give excellent results in terms of *precision* (percentage of retrieved components that are relevant) and *recall* (percentage of relevant components that are retrieved), the process of generating functional specifications is nontrivial [65]. Because these measures of distance are based on partial ordering relations, quantitative values cannot be obtained and implementing a system based on this scheme can prove to be unwieldy.

Other researchers who have proposed deduction-based approaches to software component retrieval include Zaremski and Wing [106], Penix and Alexander [81], Fischer [33], Kakeshita and Murata [44] and Luqi and Guo [59]. The central problems common to almost all deduction-based approaches are those of scalability and usability.

2.2.3 Classification-Based Techniques

Mili *et al.* [68] described a tool called *SoftClass* that uses a multi-faceted, controlled vocabulary representation. They implemented three matching algorithms: weighted Boolean retrieval, conceptual distance measures, and classification, all keyword-based. Because this is a hybrid retrieval technique, we could just as easily have discussed it under metric-based techniques. However, we have chosen to discuss it here because of the

alignment between its hierarchical key-word representation and the classification-based matching algorithm, which we briefly discuss now. The classification matching actually depends on the generalization hierarchy of the controlled set of keywords. A query matches a component if it matches the component exactly or is a specialization of the component. This is straightforward to achieve, since we are dealing with a controlled set of the keywords organized into a generalization hierarchy.

Cybulski and Reed [18] described an approach to representing requirements specification by means of a weighted multi-faceted classification scheme, whereby a predefined set of facets are defined and each requirement specification is analyzed to obtain its faceted descriptors (or terms for each facet). An advantage of this scheme is that requirements can be compared easily both with other requirements and some other artifacts, provided they are also represented using the same faceted classification. The trouble with it lies in the tedium involved in obtaining sufficiently expressive descriptors. Furthermore, the automated term extractions are not very reliable [89], which brings to question the practicality of the scheme.

Pozewaunig and Mittermeir [82] presented an approach whereby software components are clustered using their generalized signatures and then using decision trees based on test data to classify the components within each

cluster. Generalized signatures are generated by replacing types by their generalization (e.g. types double and long are replaced by type number). Retrieval is conducted by navigating through the decision tree of the appropriate cluster interactively.

Meling *et al.* [62] proposed a classification tree, which may be used to describe software artifacts. A prototype tool was implemented based on this classification tree and several software components indexed using the tree, however, Meling *et al.* have not done much on matching component descriptors based on the tree.

Czarnecki *et al.* [21] described a tool called ClassExpert that uses faceted classification technique with a controlled vocabulary. The controlled vocabulary is organized into a taxonomy (or generalization hierarchy), which is used to obtain *generalized matches*. Besides generalized match, ClassExpert can also do exact and partial match. Exact match implies that the query must be matched exactly, while partial match suggests that only part of the query need be satisfied. Generalized match means that a query may be matched with a component as long as its attribute values are a generalization of the query's.

2.3 The Unified Modeling Language (UML)

Since our work focuses primarily on object-oriented models represented in the Unified Modeling Language (UML), this section briefly introduces the UML.

In the early 1990s, the methods of Grady Booch and James Rumbaugh became very popular. A well-defined method (or software development method) would have a set of well-defined core modeling abstractions, an expressive notation for describing software development artifacts in the target domain(s), a software development process, and a set of metrics to measure the progress of development activities [23]. The Rumbaugh method, *Object Modeling Technique* (OMT) was more structure-oriented, while Booch covered the commercial and technical areas, including time-critical applications. In 1995, Booch and Rumbaugh began to combine their methods, first in the form of a common notation, to create the *Unified Method* (UM). This was soon renamed the *Unified Modeling Language* (UML), which was actually a more appropriate name, since what is being unified is in fact the notation for expressing object-oriented models. Methods are a different issue, altogether. Soon, Ivar Jacobson joined in, bringing in *use-cases* from his *Object-Oriented Software Engineering* (OOSE or simply Objectory) method, into the UML. The trio is popularly known as the

Amigos. In 1997, the Object Management Group (OMG) accepted the UML Version 1.1 as an OMG standard. Versions 1.2, 1.3, and 1.4 soon followed [75]. The OMG is an international organization supported by over 800 members, including information system vendors, software developers and vendors [77][78]. Since the middle of 2001, work has begun on the next version of the UML, version 2.0. OMG promises it will be a major upgrade [76].

The UML is a standard graphical notation for expressing a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components [77]. The UML allows expressing these things at varying levels of abstraction. Thus, the UML is expressive enough for documenting analysis- as well as design-stage models.

The official specification of the UML is contained in an OMG document titled, *OMG Unified Language Specification*. This discussion pertains to the latest version of the document at the time of writing, Version 1.4. It is made up of six chapters and two appendices. The chapters are titled as follows:

1. UML Summary
2. UML Semantics

3. UML Notation Guide
4. UML Example Profiles
5. UML Model Interchange
6. Object Constraint Language Specification

The chapters that are particularly relevant to this thesis are Chapters 2 and 5. Chapter 2 defines the UML *metamodel* using a four-layer metamodeling architecture, after the fashion of the OMG Meta-Object Facility (MOF). These four layers are described in the following table, obtained from the UML Specification [77].

Table 2: Four-layer Metamodeling Architecture [77]

Layer	Description	Example
Meta-metamodel	The infrastructure for a metamodeling architecture. Defines the language for specifying a metamodel.	<i>MetaClass, MetaAttribute, MetaOperation</i>
Metamodel	An instance of a metamodel. Defines the language for specifying a model.	<i>Class, Attribute, Operation, Component</i>
Model	An instance of a metamodel. Defines a language to describe an information domain.	<i>StockShare, askPrice, sellLimitOrder, StockQuoteServer</i>
user objects (user data)	An instance of a model. Defines a specific information domain.	<i><Acme_SW_Share_98789>, 654.56, sell_limit_order, <Stock_Quote_Svr_32123></i>

The meta-modeling layer forms the foundation for the metamodeling architecture. This layer defines the language for specifying a metamodel. A meta-metamodel defines a model at a higher level of abstraction than a metamodel layer. For example, Class in the metamodel is an instance of MetaClass in the meta-metamodel. The same can also be said between the metamodel layer and the model layer, and between the model layer and the user objects layer. The definition of the UML is actually at the metamodel layer. Thus, the UML defines a language for use at the model layer. Since the UML can be used to model entities at different layers of abstraction, the UML can be used to define itself. Thus, in the succeeding chapters, relevant portions of the UML metamodel [77], expressed in the UML, shall be used to bootstrap the discussion.

While the UML notations have been standardized since 1997, the storage format for representing UML-based models in persistent storage has not been as lucky. Every UML-supporting CASE tool has come up with own proprietary formats. It was only recently that the UML community has agreed on a standard storage format for the UML. Although Chapter 5 of the OMG UML Specification briefly addresses the issue of an interchange format, it also points to the OMG XMI Specification for further details.

The market is glutted with UML modeling tools such as Rational Rose, Together J, System Architect, Visio, Microsoft Visual Modeler, Advanced Tech GD-Pro, Visual UML, Object Domain, Object Team, etc. Unfortunately, each tool has its own proprietary storage format. This has made it difficult to interchange models across tools. Rather than building a polynomial number of bridges from and to every tool existing, a single standard interchange format for the UML will cut the number of bridges required to a single one per tool. The standard interchange format adopted for the UML is the XML Model Interchange (XMI) format. Almost all CASE tool vendors now have support for XMI in their products.

XMI (XML Metadata Interchange) is the OMG's adopted technology for interchanging models in a serialized form [78]. XMI version 1.1 was formally adopted by the OMG in February 2000. XMI focuses on the interchange of MOF (Meta Object Facility) metadata; that is, metadata conforming to a MOF metamodel.

XMI is based on the W3C's eXtensible Markup Language (XML) and has two major components:

1. The XML DTD Production Rules for producing XML Document Type Definitions (DTDs) for XMI encoded metadata. XMI DTDs

serve as syntax specifications for XML documents, and allow generic XML tools to be used to compose and validate XMI documents.

2. The XML Document Production Rules for encoding metadata into an XML compatible format. The production rules can be applied in reverse to decode XMI documents and reconstruct the metadata.

In this work, we are primarily concerned with XMI documents that obey the UML DTD [78]. For our implementation, we have specifically used UML 1.3 DTD and XMI 1.0 because at the time we were doing the implementation, these were the versions mostly supported by accessible CASE tools, such as Together and Rational Rose.

In the next chapter, we present the conceptual framework for our work.

Chapter 3

Conceptual Framework

3.1 Introduction

In this chapter, we discuss some fundamental concepts that underlie our work. Section 3.2 will introduce the representational viewpoint-oriented approach for software modeling [32]. Section 3.3 presents some assumptions we have made in this work. Section 3.4 discusses our metric composition approach. Sometimes when a set of similarity metrics, each focusing on a specific view (or aspect) of a software artifact, are composed together, these metrics might report conflicting instantiations. On the one hand, with the same instantiations, one metric might report a very high similarity value, while another might report low similarity. On the other hand, one metric might report a very high similarity and at the same time, another metric might report a very high similarity value too, but with different instantiations. A proposal to compensate for such conflicts is discussed in Section 3.5.

3.2 Multiple Views of Software

A well-known principle of modeling is the representational viewpoint-oriented approach [32][50][96]. This approach is succinctly echoed in the UML Specification as follows [77]:

Every complex system is best approached
through a small set of nearly independent
views of a model. No single view is sufficient.

For this reason, the UML was designed to support multiple views *viz.*: use-case view, structural view, behavioral view, and implementation view. Each view may be expressed by one or more diagrams. The use-case view can be expressed using use-case diagrams; the structural view can be expressed with class diagrams, the behavioral view can be expressed using a subset of state-chart, activity, sequence and collaboration diagrams; and the implementation view can be expressed with component and deployment diagrams [77].

Our survey shows that existing approaches to matching software artifacts especially, analysis and design models, have often focused only on a single view at the expense of other views (refer to Chapter 2). For instance, Blok and Cybulski [10] have focused mainly on the behavioral view and Sarireta and Vaucher [91] have only considered the class model. Our framework

differs from all these approaches by capturing multiple views in its similarity assessment.

3.3 Fundamental Assumptions

The basic assumptions underlying our approach can be expressed as follows:

Let S_1 and S_2 be two software artifacts. Suppose that each artifact S_i has n views viz. V_1, V_2, \dots, V_n . We denote view V_j of artifact S_i by the notation $V_j(S_i)$. Further, we denote the similarity between artifact S_j and S_k by $sim(S_j, S_k)$ and the similarity between views $V_i(S_j)$ and $V_i(S_k)$ by $sim(V_i(S_j), V_i(S_k))$. Similarly, we denote the dissimilarity between views $V_i(S_j)$ and $V_i(S_k)$ by $diff(V_i(S_j), V_i(S_k))$.

We assume the following properties to hold:

- $sim(S_j, S_k) = sim(S_k, S_j)$
- $sim(V(S_j), V(S_k)) = sim(V(S_k), V(S_j))$
- $diff(V(S_j), V(S_k)) = diff(V(S_k), V(S_j))$
- $sim(S_j, S_k) = \sum_i^n (\alpha_i sim(V_i(S_j), V_i(S_k)) - \beta_i diff(V_i(S_j), V_i(S_k)))$

The coefficients α_i and β_i are weights assigned to the similarities and differences of the various views. In other word, the similarity of a complex

entity can be obtained by aggregating the similarities of its parts less their differences. Studies in the domain of cognitive psychology have also made similar assumptions (*cf.* the contrast model of Tversky [98]). However, Amos Tversky's contrast model assumes similarity not to be commutative. While this is arguably true, we shall stay with this assumption for simplicity's sake. In fact, subsequently we shall assume β_i to be zeros for the same reason. Further, we are of the opinion that the differences between artifacts within the same view are less critical to the overall similarity of software artifacts. Instead, we regard the consistency in the similarities reported across the different views to be more significant. We have thus devised a measure, which we have termed *inconsistency penalty*. Suppose we have a structural similarity metric M , which reports that a certain class C_1 has the highest similarity to another class C_2 . Suppose also that another metric say a behavioral similarity metric M' (i.e. based on a behavioral view) reports C_1 's similarity to some other class, C_3 to be higher than to C_2 . Then, we say that the metrics, M and M' , have reported an inconsistency. The inconsistency penalty is a measure of the degree of inconsistency in component similarity as observed by the similarity metrics for the different views. These ideas will become clearer as we further explain them in subsequent sections of this chapter.

3.4 Metric Composition and Cascading

In this section we present a schema for composing higher-level metrics from lower-level ones. Fundamental to this framework is the concept of views (also called viewpoints or perspectives [73]) of a software model. The following list and Figure 2 below give the different views of a UML model and the diagrams that the UML provides in each view.

1. use case diagram (use-case view)
2. static diagrams (structural view)
 - a. class diagram
 - b. object diagram
3. behavior diagrams (behavioral view):
 - a. state-chart diagram
 - b. activity diagram
 - c. interaction diagrams:
 - i. sequence diagram
 - ii. collaboration diagram
4. implementation diagrams (implementation view):
 - a. component diagram
 - b. deployment diagram

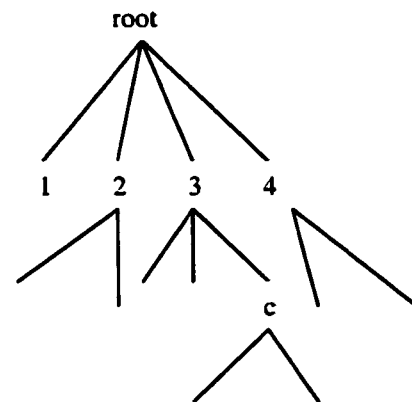


Figure 2: Tree Structure of Diagram Categories

One way to approach the problem of similarity assessment is to devise a similarity metric (or a set of similarity metrics) for at least one diagram from each view or from the most vital views depending on our specific goal. Then using the tree structure above or a scaled down version of it, we can generate an overall similarity metrics as the product of the inconsistency penalty, I , and a linear combination of the form shown in Equation 9 below:

$$sim(A, B) = I \cdot \sum_i \alpha_i \cdot sim(V_i(A), V_i(B))$$

9

I denotes the inconsistency penalty discussed in Section 3.5. V_i and sim represent views and the aggregate similarity metric as described previously. Moreover, α_i represents a weight attached to that metric, as previously described. We shall refer to this way of combining metrics from different views as *composition*.

However, when combining metrics within the same view, we shall piggyback them one on top of the other (as shown in Figure 3 below). Each metric will refine its predecessor and be refined by its successor. Metrics may be ordered in the pipeline by increasing algorithmic complexity. We shall refer to this model of combining metrics as *cascading*. Thus, with cascading and composition, a vast number of different combinational models of computing similarity measures are possible.

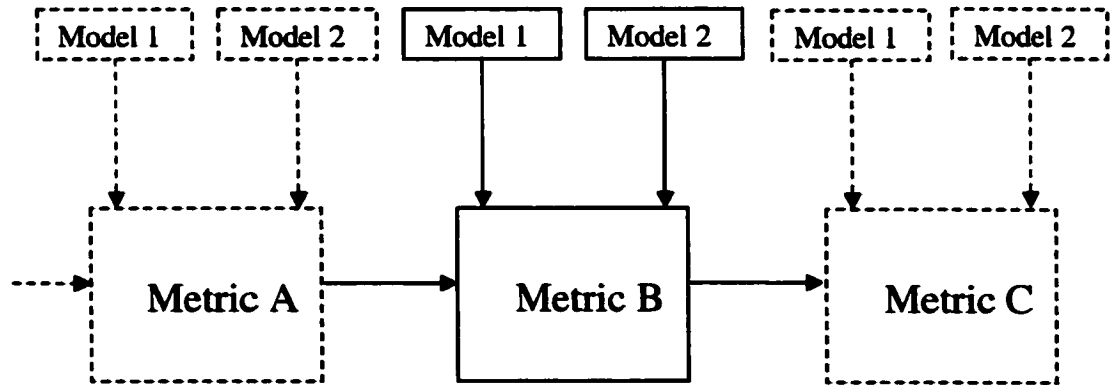


Figure 3: Metric Cascading

Due to time limitation, in this work, we focus only on the structural (static) view, while discussing a general framework that incorporates multiple views.

Our approach to matching the class view consists in cascading different measures of similarity. These measures form the subject of the next chapter.

The next section presents an approach to obtaining the *inconsistency penalty*, introduced earlier in the chapter.

3.5 Computing the Inconsistency Penalty

Our approach consists in attaching a metric (which in turn may be simple or cascaded) to each view of interest. Since these view-centric metrics are computed for each view independent of the other views, we would expect that the mappings (i.e. instantiations) associated with each metric be consistent with those of other metrics for true similarity between models. For instance, suppose that class similarity measure obtained from comparing models A and B, produces a set of mappings including the mapping $A.method1 \rightarrow B.method3$, while the sequence diagram similarity measure produces a different mapping, say $A.method1 \rightarrow B.method4$. Then, we say that these mappings are inconsistent and thus, the similarity measures cannot both be correct.

This consistency constraint is desirable for a linear combination of the metrics to be meaningful. Thus, in cases where this constraint is violated as in the above example, we try to reflect the impact of this inconsistency on the overall similarity assessment, by reducing the contribution of the concerned mapping to the overall similarity measure. We do this by multiplying the measure associated with the mappings by an adjustment factor, which we call an *inconsistency penalty* factor. The inconsistency penalty, I , may be computed as follows:

$$I = 1 - \frac{|B_1 \otimes B_2|}{|B_1 + B_2|}$$

where B_1 and B_2 are binary matrices representing the mappings according to metrics M_1 and M_2 respectively.

Operators \otimes and $+$ are the XOR and OR logical operators.

The next chapter discusses fundamental similarity metrics for the class view as well as some related design and implementation issues.

Chapter 4

Structural Similarity Metrics

4.1 Introduction

In this chapter, we discuss different similarity metrics for assessing the similarity between a pair of UML models based on information gleaned from their class diagrams. Algorithmic as well as architectural issues relating to our implementation of these metrics are also presented.

4.2 Algorithms for Class Model

One approach to comparing a pair of UML models is to use the semantics of the terms that appear in the model such as class names, attribute names, method names in a class model [88]. For this approach, we have devised two metrics: shallow semantic similarity metric and deep semantic similarity metric. The former uses only the class names of the classes in the models to be compared to compute their similarity, while the latter uses attribute and method names instead. These two metrics are discussed in Section 4.2.2 and 4.2.3 respectively. Another approach is based on comparing the signatures of the classes involved. This approach underscores our signature-matching metric, discussed in section 4.2.4 below. Yet a third approach is to use the

relationships among the classes of a class model as the criteria for comparison of the models to be compared. The relationship-based metric is the subject of Section 4.2.5. Because the first two metrics are based on an underlying semantic distance measure, we present in Section 4.2.1 below the rationale behind our choice of a semantic distance measure.

4.2.1 Semantic Distance Measure

In this section, we discuss the details of the semantic distance measure that we have chosen and why we have chosen it from the pack of options available.

The literature of Artificial Intelligence and computational linguistics is replete with approaches to measure semantic similarity (or more generally, semantic relatedness). Semantic relatedness is a more general concept in the sense that it refers to whether two concepts are related in some way, not necessarily via similarity. For instance, relatedness subsumes such relationships as meronymy, hyponymy, etc. Budanitsky and Hirst [14] identified three categories of approaches to measuring semantic relatedness found in the literature: *analytic approaches*, *statistical and machine learning approaches*, and *hybrid approaches*.

Analytic approaches attempt to use physical properties of an ontological network of concepts such as the WordNet online dictionary to estimate the

similarity between concepts. Statistical and machine learning approaches attempt to use stored-up or learned statistics as the basis of judging similarity among concepts. Many of these methods are based on Shannon's Information Theory. Hybrid approaches combine both approaches to have some of the most effective means of assessing semantic similarity and semantic relatedness.

Table 3: Table of features for different WordNet-based similarity measures

Metric	Nouns	Verbs	Adjectives	Adverbs	Value Range¹
Jiang & Conrath [42]	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	0..1
Lin [57]	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	0..1
Hirst & St-Onge [41] [97][15]	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	0..24
Leacock & Chodorow [54]	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	0..3.47
Resnik [85]	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	0.. unbounded

In our work, like Budanitsky and Hirst [14], we have limited the scope of our search for semantic distance measures to those that make use of the WordNet network as their knowledge source. This is due primarily to WordNet's opensourceness and richness of supporting tools. Table 3 above lists the five

¹ These value ranges were obtained by experimenting with the Distance-0.11 tool of Pedersen and Patwardhan [80], which implements these metrics. The author is aware that these ranges might vary from one implementation to the other, depending on the values chosen for certain parameters in the underlying algorithms.

metrics evaluated by Budanitsky and Hirst [14]. The metrics are compared based on the parts of speech that they support and the range of values that they return.

We would prefer our choice metric to return values in the unit interval and to support comparison of verbs and nouns. A unit interval makes it easy to have a normalized measure from our metric. Support for nouns and verbs, is important since modelers typically form the names of concepts they model out of these two parts of speech most of the time. While none of the metrics on Table 3 above satisfies both requirements, the Hirst and St-Onge (HSO) measure supports comparing nouns and verbs and it can be easily scaled onto the unit interval. Thus, we have chosen HSO as the underlying semantic distance metric in our computations. HSO is one of those metrics that belong in the first category (analytic methods). Fortunately, Pedersen and Patwardhan [80] recently posted a free, open-source Perl implementation of these metrics in the public domain. By doing minimal modification to the script, namely to recast it as a socket server, we have built our tool to use their script for the semantic similarity part of our work.

4.2.2 Shallow Semantic Similarity Metric

In the context of class models, we refer to a similarity metric as shallow if it does not use information within a class (such as its methods and attributes) for similarity assessment. We refer to a similarity metric as deep if it uses these types of information for its similarity assessment. In this section, we shall discuss a shallow similarity metric that uses only the semantics of words that appear in the class names of a UML model for its similarity assessment.

The Shallow Semantic Similarity Metric (SSSM) is computed according to Equation 10 below. Given a pair of models M_1 and M_2 , SSSM is computed as the average maximum similarity between the pair of classes C_i and C_j , where the former is from M_1 and the latter from M_2 respectively.

$$SSSM(M_1, M_2) = \begin{cases} 1 & \text{if } \max(|M_1|, |M_2|) = 0 \\ \frac{1}{\max(|M_1|, |M_2|)} \sum_i \max_j \left\{ \begin{matrix} hso(name(C_i(M_1)), \\ name(C_j(M_2))) \end{matrix} \right\} & \text{if } \max(|M_1|, |M_2|) > 0 \end{cases} \quad 10$$

As Algorithm 1 below shows, one of the models is designated as the pivot model. The criterion for choosing the pivot model is the number of classes in the model. The model with the fewer number of classes is chosen as the pivot. The rationale for doing this is to facilitate a preconceived performance tuning, which will reduce the complexity of our algorithm slightly. Each class in the pivot model is compared with every class in the non-pivot model

using HSO. The values returned by HSO are stored in a matrix `CM` and the best matches are stored in a Boolean matrix `greedyMatrix`. To compute SSSM, the values for these *best matches* are summed up and normalized by dividing with the cardinality of the non-pivot model. We have chosen to use the non-pivot model, in order to forestall obtaining a maximum similarity value returned when a pivot model happens to be a perfect subset of the non-pivot model. Consider the case of comparing a class model containing a single class C_I with another class model with hundreds of classes, one of which happens to bear the same name as C_I . The first model will be selected as the pivot since it has fewer classes and the iteration is controlled by the cardinality, which is 1 in this case. Thus, the numerator in Equation 10 above will return 1. Had we not normalized with the cardinality of the nonpivot model, this would have given a similarity of 1 for these two models, in contrast with our intuitive judgment.

Algorithm 1: SSSM**Input:** Models M_{pivot} and $M_{nonpivot}$ **Output:** A similarity value in the interval $[0, 1]$

```

 $\forall i, j: 0 < i < |M_{pivot}|, 0 < j < |M_{nonpivot}|$ 
 $CM_{i,j} = hso(name(C_i(M_{pivot})), name(C_j(M_{nonpivot})))$ 
greedyMatrix = computeGreedyMatrix(CM)
bestMatches = applyHungarianAlgorithm(CM)
return computeMeasure(CM, greedyMatrix)

```

```

computeMeasure(CM, greedyMatrix):

```

```

 $\forall i, j: 0 < i < |M_{pivot}|, 0 < j < |M_{nonpivot}|$ 
  if (greedyMatrix[i][j])
    sum = sum + CM[i][j]
  if ( $|M_{nonpivot}| = 0$ )
    return 1.0
  else
    return  $\frac{sum}{|M_{nonpivot}|}$ 

```

The *computeGreedyMatrix()* routine in Algorithm 1 above returns a Boolean matrix that represents pairings that produce maximum-cost assignments (i.e. maximum similarity values) of classes from one model to the other with repetition. The algorithm has been called greedy because it does not consider assignments already made when making new ones. It simply assigns to each class in the pivot model the class in the nonpivot

model that has the maximum similarity values, without consideration for choices made for other classes in the pivot model. Thus, assigned classes from the nonpivot model may not be unique to each class in the pivot model.

The `applyHungarianAlgorithm()` routine is similar, but with a 1-1 constraints that ensures that no two classes from one model is assigned to the same class in the other model. As the name implies, this routine is an implementation of the Hungarian algorithm [52].

4.2.2.1 Algorithmic Complexity of SSSM

The algorithm has the complexity $O(|M_{pivot}| \cdot |M_{nonpivot}| \cdot |HSO|)$, where $|M_{pivot}|$ and $|M_{nonpivot}|$ are the number of classes (and interfaces) in models M_{pivot} and $M_{nonpivot}$ respectively, while $|HSO|$ is the algorithmic complexity of the `hso()` algorithm. The `hso()` algorithm essentially searches for the shortest path between two nodes in a semantic network (i.e. a directed graph). A well-known algorithm for this problem is Dijkstra's algorithm which has a square complexity [2]. Thus, SSSM has a time complexity of the form $O(|M_{pivot}| \cdot |M_{nonpivot}| \cdot m^2)$, where m is the number of nodes in the semantic network. Since m is constant (albeit, a very high one) and independent of the number of classes in the models under comparison, we could say that SSSM is $O(n^2)$, where $n = \max(|M_{pivot}|, |M_{nonpivot}|)$. The actual

running time of this algorithm is high because the multiplicative constant (m^2) involved is also high.

4.2.3 Deep Semantic Similarity Metric

The Deep Semantic Similarity Metric (DSSM) does not use the name of the classes in the matching. Rather it goes right into the classes and calls on HSO to match attribute names with attribute names and method names with method names.

The computation of DSSM is shown as a series of steps. First, we compute the average maximum similarity among attributes of classes from model M_1 and those of classes from model M_2 as Equation 11 below shows. Second, we compute a similar value for operations of classes from both models as shown in Equation 12. Given a pair of models, M_1 and M_2 , DSSM is computed as the average. Finally, we compute the weighted average of these two values across the two models as indicated in Equation 13 below.

$$X(C_i, C_j) = \begin{cases} 1 & \text{if } \max(|M_r|, |M_s|) = 0 \\ \frac{\sum_p \max_q (hso(A_p(C_i), A_q(C_j)))}{\max(|A_p|, |A_q|)} & \text{if } \max(|M_r|, |M_s|) > 0 \end{cases} \quad 11$$

$$Y(C_i, C_j) = \begin{cases} 1 & \text{if } \max(|M_r|, |M_s|) = 0 \\ \frac{\sum_r \max_s (hso(M_r(C_i), M_s(C_j)))}{\max(|M_r|, |M_s|)} & \text{if } \max(|M_r|, |M_s|) > 0 \end{cases} \quad 12$$

$$DSSM(M_1, M_2) = \begin{cases} 1 & \text{if } \max(|M_1|, |M_2|) = 0 \\ \frac{\sum_i \max_j \left\{ \alpha X(C_i(M_1), C_j(M_2)) \right.}{\max(|M_1|, |M_2|)} & \text{if } \max(|M_1|, |M_2|) > 0 \end{cases} \quad 13$$

As Algorithm 2 below shows, DSSM algorithm is similar to that of SSSM but it is doubly nested. Each pair of classes is matched (using `deepClassCompare` routine of Algorithm 2) by matching their attribute sets using HSO and then taking the maximums and normalizing. We do the same thing for the method sets. Then, we take a linear combination of these two. In the current implementation, we have used fixed weights for computing the linear combination. A future improvement would be to experiment with different weighting schemes and then settle for one that produces the most effective results. At the higher level, the UML model pair are matched by summing the maximums of class similarity measures (computed using `deepClassCompare`) and normalizing as before using the cardinality of the bigger model.

Algorithm 2: DSSM**Input:** Models M_{pivot} and $M_{nonpivot}$ **Output:** A similarity value in the interval $[0, 1]$ *deepSemantic :* $\forall i, j : 0 < i < |C(M_{pivot})|, 0 < j < |C(M_{nonpivot})|$ $CM_{i,j} = \text{deepCompareClasses}(C_i(M_{pivot}), C_j(M_{nonpivot}))$ $\text{greedyMatrix} = \text{computeGreedyMatrix}(CM)$ $\text{bestMatches} = \text{applyHungarianAlgorithm}(CM)$ $\text{return computeMeasure}(CM, \text{greedyMatrix})$ *deepClassCompare*(C_1, C_2): $\forall i, j : 0 < i < |A(C_1)|, 0 < j < |A(C_2)|$ $SA_{i,j} = \text{hso}(A_i(C_1), A_j(C_2))$ $\text{gAMatrix} = \text{computeGreedyMatrix}(SA)$ $\forall k, l : k < |M(C_1)|, 0 < l < |M(C_1)|$ $SM_{i,j} = \text{hso}(M_i(C_1), M_j(C_2))$ $\text{gMMatrix} = \text{computeGreedyMatrix}(SM)$ $\text{return } \alpha * \text{computeMeasure}(SA, \text{gAMatrix})$ $+ \beta * \text{computeMeasure}(SM, \text{gMMatrix})$ **4.2.3.1 Algorithmic Complexity of DSSM**

DSSM has a time complexity of the form $O(|M_{pivot}| \cdot |M_{nonpivot}| \cdot |C|^2 \cdot m^2)$, where $|C|$ denotes the number of features² in class C (the class with the largest number of features in the two models under comparison), m is the number of nodes in the semantic network. Since m is constant and

independent of the number of classes in the models under comparison, we say that DSSM is $O(n^2 \cdot |C|^2)$, where $n = \max(|M_{pivot}|, |M_{nonpivot}|)$. This algorithm is quite expensive. The concept of cascading can help improve the running time of a retrieval engine that uses this metric by placing less expensive algorithm early in the filter pipeline and then placing DSSM at the end, thus reducing the number of classes to compare using DSSM.

Besides the high cost of this algorithm, another factor that makes it less attractive is the following naming malpractices that exist among developers: shortened words (e.g. `exec` for `execute`), uncapitalized acronyms (e.g. `Xml` for `XML`), multiple words without initial capitals (e.g. `Copyfile` for `CopyFile`). These malpractices make it nontrivial to tokenize names and compare them with other names semantically.

4.2.4 Signature-Based Similarity Metric

The algorithm for this metric is very similar in structure to that of the deep semantic metric. Rather than using the names of attributes and methods, it uses their signatures for comparison. For an attribute, we define its signature simply as its type, while a method's signature includes the types of its parameters and its return type.

² A feature is an attribute or operation (i.e. method) of a class or an interface [77].

The computation of SBSM is shown as a series of steps. First, we compute the average maximum similarity among attributes of classes from model M_1 and those of classes from model M_2 as Equation 14 below shows. Second, we compute a similar value for operations of classes from both models as shown in Equation 15. Given a pair of models, M_1 and M_2 , SBSM is computed as the average. Finally, we compute the weighted average of these two values across the two models as indicated in Equation 16 below.

$$X(C_i, C_j) = \begin{cases} 1 & \text{if } \max(|M_r|, |M_s|) = 0 \\ \frac{\sum_p \max_q (\text{typesim}(A_p(C_i), A_q(C_j)))}{\max(|A_p|, |A_q|)} & \text{if } \max(|M_r|, |M_s|) > 0 \end{cases} \quad 14$$

$$Y(C_i, C_j) = \begin{cases} 1 & \text{if } \max(|M_r|, |M_s|) = 0 \\ \frac{\sum_r \max_s (\text{signsim}(M_r(C_i), M_s(C_j)))}{\max(|M_r|, |M_s|)} & \text{if } \max(|M_r|, |M_s|) > 0 \end{cases} \quad 15$$

$$SBSM(M_1, M_2) = \begin{cases} 1 & \text{if } \max(|M_1|, |M_2|) = 0 \\ \frac{\sum_i \max_j \left\{ \alpha X(C_i(M_1), C_j(M_2)) + \beta Y(C_i(M_1), C_j(M_2)) \right\}}{\max(|M_1|, |M_2|)} & \text{if } \max(|M_1|, |M_2|) > 0 \end{cases} \quad 16$$

In our current implementation, the weights are chosen as $\alpha = 0.4$ and $\beta = 0.6$ to reflect the relative complexity of attribute types and method signatures respectively. Algorithm 3 below implements this metric. It works more or less the same way as DSSM, except that here, two new methods `typesim` and `signsim` are used to match attributes and methods respectively. The

first method, `typesim` simply does a string comparison of the types, more or less, like what Zaremski and Wing [105] termed “exact match.” The second method, simply invokes `typesim` for comparing the parameter types and for the return types of the methods from both classes being compared.

Algorithm 3: SBSM**Input:** Models M_{pivot} and $M_{nonpivot}$ **Output:** A similarity value in the interval $[0, 1]$

```

 $\forall i, j: 0 < i < |C(M_{pivot})|, 0 < j < |C(M_{nonpivot})|$ 
 $CM_{i,j} = \text{signcompare}(C_i(M_{pivot}), C_j(M_2))$ 
 $\text{greedyMatrix} = \text{computeGreedyMatrix}(CM)$ 
 $\text{bestMatches} = \text{applyHungarianAlgorithm}(CM)$ 
 $\text{return computeMeasure}(CM, \text{greedyMatrix})$ 

```

```

 $\text{signcompare}(C_1, C_2):$ 
 $\forall i, j: 0 < i < |A(C_1)|, 0 < j < |A(C_2)|$ 
 $SA_{i,j} = \text{typesim}(A_i(C_1), A_j(C_2))$ 
 $\forall i, j: i < |M(C_1)|, 0 < j < |M(C_2)|$ 
 $SM_{i,j} = \text{signsim}(M_i(C_1), M_j(C_2))$ 
 $\text{gAttrMatrix} = \text{computeGreedyMatrix}(SA)$ 
 $\text{gMethodMatrix} = \text{applyHungarianAlgorithm}(SM)$ 
 $\text{return } \alpha * \text{computeMeasure}(SA, \text{gAttrMatrix})$ 
 $\quad + \beta * \text{computeMeasure}(SM, \text{gMethodMatrix})$ 

```

4.2.4.1 Algorithmic Complexity of SBSM

SBSM has a complexity of the form $O(|M_{pivot}| \cdot |M_{nonpivot}| \cdot |C|^2 \cdot m)$, where $|C|$ denotes the number of features in class C (the class with the largest number of features in the two models under comparison), and m is the complexity of $\text{signsim}()$. Since m is constant, the complexity of SBSM reduces to $O(|M_{pivot}| \cdot |M_{nonpivot}| \cdot |C|^2)$. The concept of cascading can help improve the running time of a retrieval engine that uses this metric by placing less

expensive algorithm early in the filter pipeline and then placing it towards the end, thus reducing the number of classes to compare using it.

4.2.5 Relationships-Based Similarity Metric

We define the fingerprint of a class, C , as the tuple $F_C = (G, I, S)$, where G denotes the set of direct superclasses of C ; I the set of interfaces implemented by C ; and S the set of direct subclasses of C .

The Relationships-Based Similarity Metric (RBSM) is computed, as shown in equation 17 below, as the average maximum similarity between the fingerprints of classes C_i and C_j , where the former is from M_1 and the latter from M_2 respectively.

$$RBSM(M_1, M_2) = \begin{cases} 1 & \text{if } \max(|M_1|, |M_2|) = 0 \\ \frac{1}{\max(|M_1|, |M_2|)} \sum_i \max \left\{ \frac{relsim(fingerprint(C_i(M_1)), fingerprint(C_j(M_2)))}{1} \right\} & \text{if } \max(|M_1|, |M_2|) > 0 \end{cases}$$

$$\text{where } relsim(F, F') = \alpha \frac{|G(F) \cap G(F')|}{|G(F) \cup G(F')|} + \beta \frac{|I(F) \cap I(F')|}{|I(F) \cup I(F')|} + \gamma \frac{|S(F) \cap S(F')|}{|S(F) \cup S(F')|} \quad 17$$

The symbols $G(F)$, $I(F)$, and $S(F)$ denote respectively, the sets G , I , and S in the tuple F . The weights α , β and γ add up to unity. They denote the relative importance of each type of relationship in similarity assessment. In our current implementation, the weights have been assigned equal values of $1/3$.

Algorithm 4: RBSM**Input:** Models M_{pivot} and $M_{nonpivot}$ **Output:** A similarity value in the interval [0, 1]

```

 $\forall i, j: 0 < i < |M_{pivot}|, 0 < j < |M_{nonpivot}|$ 
     $CM_{i,j} = relsim(fingerprint(C_i(M_{pivot})), fingerprint(C_j(M_{nonpivot})))$ 
     $greedyMatrix = computeGreedyMatrix(CM)$ 
     $bestMatches = applyHungarianAlgorithm(CM)$ 
    return computeMeasure(CM, greedyMatrix)

```

4.2.5.1 Algorithmic Complexity of RBSM

The RBSM algorithm has a complexity $O(|M_{pivot}| \cdot |M_{nonpivot}| \cdot m)$, where m is the complexity of `relsim()`, which depends on the size of the fingerprints of the classes in the models under comparison. The size of the fingerprint of a class is essentially the number of relationships it has with the rest of the model. This number is bounded above by the number of classes in the bigger model. In other words, $m = O(\max(|M_{pivot}|, |M_{nonpivot}|))$. Thus, RBSM is $O(n^3)$ where $n = \max(|M_{pivot}|, |M_{nonpivot}|)$. Although, this algorithm has a higher time-complexity than SSSM, the actual running time of the latter is much higher in most cases because of its high multiplicative constant.

4.3 Implementation

In our implementation, each of the four metrics discussed above is represented by a class of its own. For instance, SSSM is implemented as a class called `ShallowSemanticFilter`, DSSM by the class `DeepSemanticFilter`, SBSM by `SignatureFilter`, and RBSM by `RelationshipsFilter`. Recall from Section 3.4 above that each metric within the same modeling view is envisioned as a filter in a pipeline. This has informed the choice of the `Filter` suffix at the end of each of these class names.

Each of these classes has the following private attributes:

```
double [][] simMatrix
boolean [][] greedyMatches
boolean [][] bestMatches
```

Each of these two-dimensional arrays represents a matrix whose columns correspond to classes from one of the models under comparison, while the rows correspond to the other. The elements of the matrix, `simMatrix`, contain the class-to-class similarity value as computed by the metric concerned. The elements of `greedyMatches` and `bestMatches` contain Boolean values, indicating whether the corresponding pair of classes is considered a matching that would contribute to the computation of the

overall model-to-model similarity value. The `greedyMatches` matrix is computed using a greedy approach while the `bestMatches` matrix is computed using the Hungarian Algorithm. Both of these approaches are described above in section 4.2.2.

In order to create a loosely coupled design and because of the commonality among filters, each metric class has been made to implement the `AbstractFilter` interface, shown below:

```
public interface AbstractFilter {  
    void apply();  
    void apply(AbstractFilter filter);  
    double [][] getSimilarityMatrix();  
    boolean [][] getGreedyMatrix();  
    boolean [][] getBinaryMatrix();  
}
```

The crux of the logic implementing each metric is contained in its `apply()` method. The `apply()` method for each of the metrics has the following sequence of steps:

Step 1: Compute `simMatrix`

Step 2: Compute `bestMatches`

Step 3: Compute `greedyMatches`

Step 4: Compute overall measure using `bestMatches`

Step 5: Compute overall measure using `greedyMatches`

Each of these steps is identical for all the metrics, except Step 1. Each metric computes this matrix according to the definition of the particular metric. For instance, in the case of SSSM, each element of `simMatrix` is computed as the semantic distance between the class names of the classes corresponding to the element's row and column.

Figure 4 below further presents a pictorial view of the filter metaphor. The dotted lines indicate that a preceding filter may or may not be present. The same also goes for the succeeding filter.

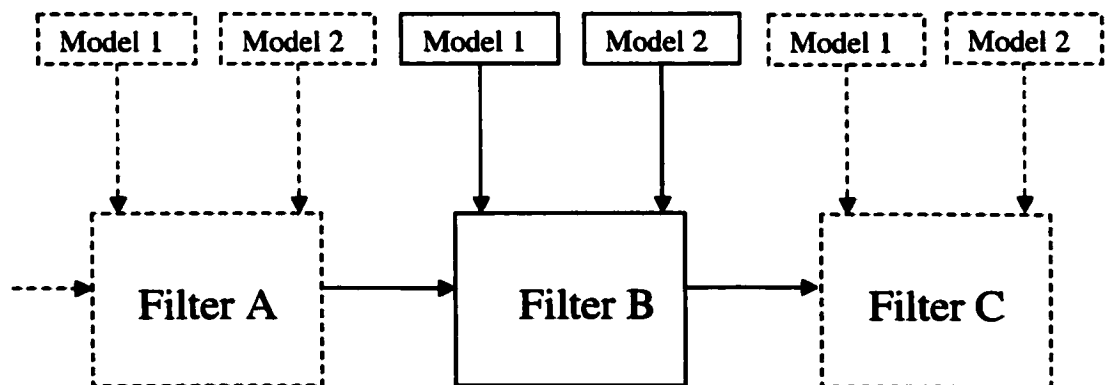


Figure 4: The Filter Metaphor

Figure 5 below shows a part of the architecture of our tool that shows how the classes that make up the metrics in the structural view are organized. These classes and two utility classes are weaved together according to the

Composite Filter Design pattern [103]. The Composite Filter pattern integrates three well-known design patterns, viz. the *Strategy* pattern [34], the *Composite* pattern [36] and the *Filter* pattern [34]. The Strategy design pattern makes it possible to deal with the different metrics the same way. The client class would deal with all the metrics the same way, irrespective of whether the metric is a cascade of several other metrics or a hierarchy of them. The hierarchy part is made possible by the Composite pattern, which can be used to compose metrics from different views into a higher-level aggregate metric in a tree-like fashion. Finally, the filter pattern is used to realize the filter metaphor as depicted in Figure 4 above. Metrics within the same view are strung together one after the other in a pipeline fashion, each filter being fed with input from the previous filter (if one exists), and itself feeding into the next filter (if one exists).

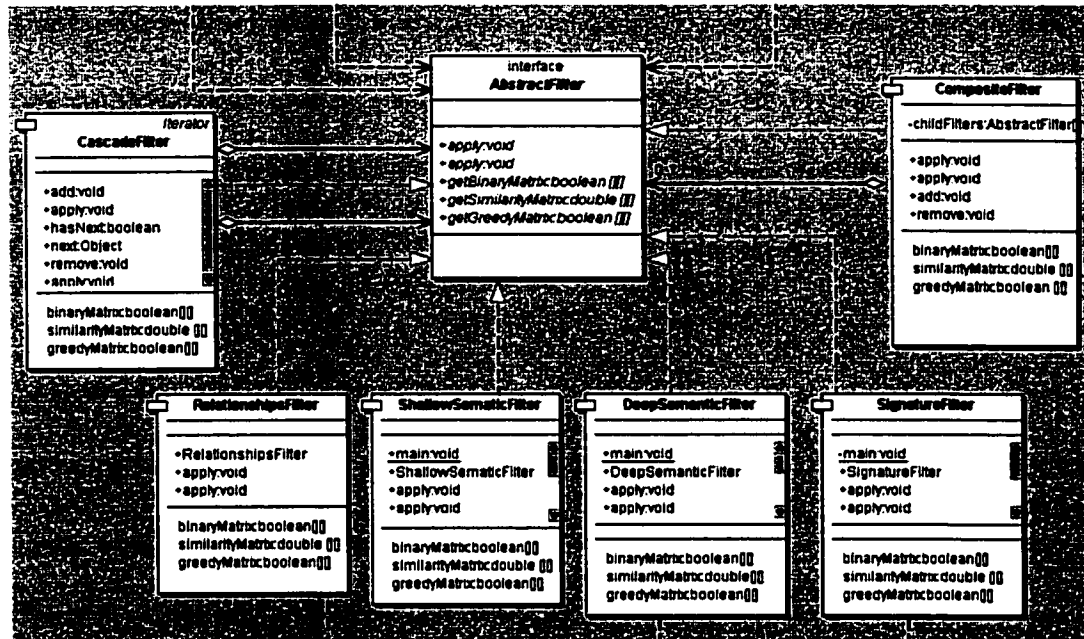


Figure 5: The Composite Filter Pattern Applied to Class Similarity Metrics

This structuring of things makes it possible to combine metrics in a variety of ways and to grow our metric set easily and seamlessly, with minimal maintenance overhead. For instance, in order to add a new metric, say one that computes the consistency of these four, we simply add a new class that implements the **AbstractFilter** interface and code the required logic, and then we can use it from the client classes just like any of the existing metrics.

In the next chapter, we present a validation of the metrics presented in this chapter. Attempt is made to validate all four metrics theoretically and empirically.

Chapter 5

Evaluation

5.1 Introduction

In this chapter, we present an evaluation of our metrics. We present two kinds of evaluation. In Section 5.2, we present a theoretical validation of the metrics using a set of axioms[57][90][88][95][42]. These axioms are actually features that we expect a valid similarity metric to have and most of them are based on assumptions that agree with our intuitive understanding of similarity. Section 5.3 presents an empirical validation of our work, using case studies to practically test if our metrics agrees with our intuition.

5.2 Theoretical Validation

Traditionally, similarity metrics (or more precisely distance metrics) have often been analytically verified using the metric axioms [57][90][88][95][42]. These axioms can be stated as follows (adapted slightly to conform to a similarity function):

1. **Maximum self-similarity:** For all classes C_1, C_2 , $\text{sim}(C_1, C_1) \geq \text{sim}(C_1, C_2)$.

2. Symmetry: For all classes C_1, C_2 , $sim(C_1, C_2) = sim(C_2, C_1)$.
3. Monotonicity[42]: The similarity will monotonically increase when there is a common part added to the two objects being compared.
4. Triangular Inequality: For all classes C_1, C_2, C_3 . Let $diff(C_1, C_2)$ denote the structural distance between classes C_1 and C_2 , expressed as: $diff(C_1, C_2) = 1 - sim(C_1, C_2)$. Then this property states that the distance function $diff$, satisfies the inequality:

$$diff(C_1, C_3) \leq diff(C_1, C_2) + diff(C_2, C_3).$$

Our metrics can be shown to satisfy the first three axioms, viz. self-similarity, maximality, and symmetry. The fourth (triangular inequality) has been shown not to be such a very important property [57], so we have ignored it in the following discussions.

5.2.1 Theoretical Validation of SSSM

Maximum Self-similarity: There are essentially two cases to consider. The first case is when two trivial models (i.e. two empty models) are to be compared. In that case, SSSM (reproduced below for readability) evaluates to 1. The other case is when two identical non-trivial models are to be compared. In that case, $hso()$ will always return 1 and thus the whole expression evaluates to 1. In all other cases, SSSM will always return a value

less than 1, since the numerator in Equation 18 below will then always be less than the denominator.

$$SSSM(M_1, M_2) = \begin{cases} 1 & \text{if } \max(|M_1|, |M_2|) = 0 \\ \frac{1}{\max(|M_1|, |M_2|)} \sum_i \max_j \left\{ \begin{array}{l} hso(name(C_i(M_1)), \\ name(C_j(M_2))) \end{array} \right\} & \text{if } \max(|M_1|, |M_2|) > 0 \end{cases} \quad 18$$

Symmetry: To show this, it suffices to show that $hso()$ is symmetric. There is experimental support for this.

Monotonicity: As can be inferred from Equation 18 above, when common components are added to the two models under comparison, both the numerator and denominator of the right hand side (RHS) expression increase equally, causing their ratio (SSSM) to also increase, unless it is already equal to unity. In that case, it remains the same. In other words, SSSM has a direct positive correlation with the number of common components between the two models.

5.2.2 Theoretical Validation of DSSM

Maximum Self-similarity: There are essentially two cases to consider. The first case is when two trivial models (i.e. two empty models) are to be compared. In that case, DSSM (reproduced below for readability) evaluates to 1. The other case is when two identical non-trivial models are to be compared. In that case, expression in within braces in Equation 19 below

will always return 1 and thus the whole expression evaluates to 1. In all other cases, the similarity value returned by DSSM will always be less than 1. This can easily be seen in Equation 19 by observing that the bracketed expression in the numerator of the lower part of Equation 19 will always result in a value less than 1.

$$DSSM(M_1, M_2) = \begin{cases} 1 & \text{if } \max(|M_1|, |M_2|) = 0 \\ \frac{\sum_i \max_j \left\{ \alpha X(C_i(M_1), C_j(M_2)) \right.}{\max(|M_1|, |M_2|)} & \text{if } \max(|M_1|, |M_2|) > 0 \end{cases} \quad 19$$

Symmetry: To show this, it suffices to show that the both functions X and Y in Equation 19 are symmetric. An observation of the definition of these functions (in Equations 11 and 12 respectively, shows that indeed that we only need to show that $hso()$ is symmetric (*cf.* Section 5.2.1 above).

Monotonicity: When more of common components are added to the two models under comparison in Equation 19 above, both the numerator and denominator of the right hand side (RHS) expression increase equally, causing their ratio (DSSM) to also increase, unless it is already equal to unity. In that case, it stays the same. (*cf.* 5.2.1 above.)

5.2.3 Theoretical Validation of SBSM

Maximum Self-similarity: There are essentially two cases to consider. The first case is when two trivial models (i.e. two empty models) are to be compared. In that case, SBSM (reproduced below for readability) evaluates to 1. The other case is when two identical non-trivial models are to be compared. In that case, the numerator will always return 1 and thus the whole expression evaluates to 1.

$$SBSM(M_1, M_2) = \begin{cases} 1 & \text{if } \max(|M_1|, |M_2|) = 0 \\ \frac{\sum_i \max_j \left\{ \alpha X(C_i(M_1), C_j(M_2)) + \beta Y(C_i(M_1), C_j(M_2)) \right\}}{\max(|M_1|, |M_2|)} & \text{if } \max(|M_1|, |M_2|) > 0 \end{cases} \quad 20$$

Symmetry: To show this, it suffices to show that the both functions X and Y in Equation 20 are symmetric. An observation of the definition of these functions (in Equations 14 and 15 respectively, shows that indeed that we only need to show that *typesim()* and *signsim()* are symmetric.

Monotonicity: It can be inferred from Equation 20 above that as the common components between the two models under comparison increase, both the numerator and denominator of the right hand side (RHS) expression increase equally, causing their ratio (SBSM) to also increase, unless it is already equal to unity, in which case, it stays constant.

5.2.4 Theoretical Validation of RBSM

Maximum Self-similarity: There are essentially two cases to consider. The first case is when two trivial models (i.e. two empty models) are to be compared. In that case, RBSM (reproduced below for readability) evaluates to 1. The other case is when two identical non-trivial models are to be compared. In that case, *relsim()* will always return 1 and thus the whole expression evaluates to 1.

$$RBSM(M_1, M_2) = \begin{cases} 1 & \text{if } \max(|M_1|, |M_2|) = 0 \\ \frac{1}{\max(|M_1|, |M_2|)} \sum_i \max \left\{ \frac{relsim(fingerprint(C_i(M_1)), fingerprint(C_j(M_2)))}{fingerprint(C_j(M_2))} \right\} & \text{if } \max(|M_1|, |M_2|) > 0 \end{cases} \quad 21$$

Symmetry: To show this, it suffices to show that *relsim()* is symmetric. Since fingerprints are set and the *relsim()* function is defined in set-theoretic terms, it follows logically that *relsim()* is symmetric.

Monotonicity: It can be inferred from Equation 21 above that, as the common components between the two models under comparison increase, both the numerator and denominator of the right hand side (RHS) expression increase equally, causing their ratio (SBSM) would also increase, unless it is already equal to unity. In that case, it stays the same.

5.3 Empirical Validation

In this section, we present an empirical validation of our metrics. Knowing the difficulty of this task [5], we adopt the characterization scheme of Lott and Rombach [58], as recommended by Miller [70]. Before getting into the validation itself, we first introduce the representational theory of measurement.

Measurement is defined as a mapping from the empirical world to the formal, relational world [31]. This means that measurement usually starts out as a set of intuitive judgments and opinions that we make about things. For instance, we have a notion of height and can tell when one person is taller than another is, but we most often cannot put a figure to the height of either of them. Our intuition is not precise enough to do that. In order to be able to, we need a more sophisticated measurement system. Whatever measurement system we adopt must agree with our intuitive understanding and judgments about the entities involved. This condition is called the *representation condition*. The representation condition lies at the heart of the representational theory of measurement. While there are other theories of measurement, the representational theory of measurement has attracted the most attention in the software engineering community [31][30].

According to Fenton [30], “*Validating a software measure in the assessment sense is equivalent to demonstrating empirically that the representation condition is satisfied for the attribute being measured.*” Thus in the next section, we attempt to show that our similarity measures preserve our intuition about similarity by conducting a series of case studies using UML models of pieces of software that are easily accessible so that our study can be repeatable and independently verified by other researchers.

5.3.1 Our Intuition about Similarity

The following list presents our intuition about the similarity between a pair of UML models. We note the equivalence between this list and the first three axioms presented in Section 5.2 above. They have been repeated here for easier reference in the succeeding discussions.

Intuition 1: Maximum similarity is reached when a model is compared with itself (*cf.* Axiom 1 in Section 5.2).

Intuition 2: If A is similar to B by some degree, then B is also similar to A by the same degree (*cf.* Axiom 2 in Section 5.2).

Intuition 3: Similarity grows with commonality and reduces with difference. The more models have in common, the greater their similarity. The more differences they have the smaller their

similarity (cf. Axiom 3 in Section 5.2).

Intuition 1 suggests that if we take the UML model of a version of a software product, and compare it with itself using a similarity metric, we should obtain a 100% similarity. Intuition 2 suggests that a similarity metric should be symmetric. Intuition 3 suggests that given a product with multiple versions, we would expect a good similarity metrics would report higher similarity values for versions that are closer to each other and lower values for versions that are farther apart.

5.3.2 Goals, Hypothesis and Theories

The goal of this empirical validation can be stated as follows:

To analyze our metrics (SSSM, DSSM, SBSM and RBSM) for assessing their effectiveness at matching UML models from the viewpoint of the software architect in the context of software reuse.

5.3.2.1 Hypotheses

The hypotheses to test can be stated as follows:

- H_{1, SSSM}:** SSSM satisfies intuition 1 in Section 5.3.1 above.
- H_{2, SSSM}:** SSSM satisfies intuition 2 in Section 5.3.1 above.
- H_{3, SSSM}:** SSSM satisfies intuition 3 in Section 5.3.1 above.

- H_{1, DSSM}:** DSSM satisfies intuition 1 in Section 5.3.1 above.
- H_{2, DSSM}:** DSSM satisfies intuition 2 in Section 5.3.1 above.

H_{3, DSSM}: DSSM satisfies intuition 3 in Section 5.3.1 above.

H_{1, SBSM}: SBSM satisfies intuition 1 in Section 5.3.1 above.

H_{2, SBSM}: SBSM satisfies intuition 2 in Section 5.3.1 above.

H_{3, SBSM}: SBSM satisfies intuition 3 in Section 5.3.1 above.

H_{1, RBSM}: RBSM satisfies intuition 1 in Section 5.3.1 above.

H_{2, RBSM}: RBSM satisfies intuition 2 in Section 5.3.1 above.

H_{3, RBSM}: RBSM satisfies intuition 3 in Section 5.3.1 above.

5.3.2.2 Theories

Each of the following is an aspect of a class model that contributes to class-model similarity assessment [88]: signature, class name semantics, semantics of attribute and method names, and relationships. Thus, each one of them can be used in a similarity assessment.

5.3.3 Experiment Plan

Owing to the difficulty of obtaining UML models, our selection of experimental objects could not be randomized. However, we have used CASE tools to generate UML models of software that are freely available in the public domain in the interest of repeatability.

5.3.3.1 Experimental Design

In order to test hypotheses $H_{1,SSSM}$, $H_{1,DSSM}$, $H_{1,SBSM}$, and $H_{1,RBSM}$, we shall compare each model in our set of experimental objects to itself and compute the percentage of times when each metric results in a value of 1.

To test hypotheses $H_{2,SSSM}$, $H_{2,DSSM}$, $H_{2,SBSM}$, and $H_{2,RBSM}$, we shall exhaustively compare every pair of models using SSSM, DSSM SBSM and RBSM respectively.

In order to test hypotheses $H_{3,SSSM}$, $H_{3,DSSM}$, $H_{3,SBSM}$, and $H_{3,RBSM}$, we adopt a the partial factorial design shown in Table 4 below, where $Model_i$ denotes the UML model of the i -th version of a piece of software. The assumption is that as we move down the $Model_1$ column, the difference between the pair of models increases, since later versions usually preserve some of the existing code and add some additional code as well. As we move down the $Model_n$ column, however, we assume that the common code between the pairs of models being compared is increasing gradually, since later versions tend to have more in common with the latest version than earlier ones.

Table 4: Experimental Design for H₃.

Model Name	Model Size	Model_1				Model_n			
		Intuitive Rank	<Metric>		<Metric> Rank	Intuitive Rank	<Metric>		<Metric> Rank
			Greedy	1-1			Greedy	1-1	
Model_1									
Model_2									
Model_3									
...									
Model_n									

Each similarity metric is used to measure the similarity between *Model₁* and each of the other models. The same is also done with *Model_n*. The intuitive rankings are recorded into the *Intuitive Rank* column, raw measurements <Metric> and the corresponding ranks are entered into the <Metric> and <Metric> Rank columns respectively.

5.3.3.2 Treatments (Similarity Metrics)

The treatments that are applied to the experimental objects (namely, pairs of UML models) are the four UML model similarity metrics discussed in the previous chapter, viz. SSSM, DSSM, SBSM, and RBSM.

5.3.3.3 Objects

The experimental objects used include the following:

1. The java.lang package of JDK1.1 through J2SDK1.4
2. Apache ANT 1.1 through Apache ANT 1.5

The jar files for each of the products is first reverse-engineered into a rose model using Rational Rose 2000. The rose models are then exported as XMI 1.0 for UML 1.3 files using Together J 5.0, which form the input to our UML Model Comparison Tool.

5.3.3.4 Subjects

Treatment application is completely automated, so effects of subject bias and differences are not a consideration.

5.3.3.5 Data Collection and Validation Procedures

Results are logged automatically to a text file by the program. Extensive testing had earlier been carried out using small UML models. Bugs discovered in the process were fixed.

5.3.3.6 Data Analysis Procedures

With hypotheses $H_{1,**SM}$ and $H_{2,**SM}$, we compute the average number of cases on which the associated intuition is satisfied. Thus, our test statistic is the arithmetic mean. The rejection criteria will be:

$$\text{Reject } H_{1,**SM}, H_{2,**SM} \text{ if } \bar{x} < .95$$

In the context of hypotheses $H_{3,**SM}$, we attempt to obtain a measure of correlation between the intuitive ranking and the ranking generated by applying our metrics. Spearman rank correlation coefficient [63] will be computed for each case.

Spearman's nonparametric test for rank correlation [63] will be used to test the hypotheses $H_{3,**SM}$ stated in the Section 5.3.2.1 above at the significance level of $\alpha = 0.01$ (i.e. we have a 99% confidence in our conclusions). Our test statistic is the Spearman rank correlation coefficient, r_s .

$$r_s = 1 - \frac{6 \sum_{i=1}^n d_i^2}{n(n^2 - 1)} \quad 22$$

where d_i is the difference between the ranks of the two ranks of the i -th observation. A hypothesis is rejected if r_s (Spearman's rank correlation coefficient computed for the sample) is less than the estimated population

rank correlation coefficient at the specified significance level, r_α . The Rejection criteria for hypotheses $H_{3,**SM}$ can be stated as follows:

$$\text{Reject } H_{3,**SM}, \text{ if } r_s \leq r_\alpha$$

Estimates of the population's rank correlation coefficient can be obtained from a statistical table for small sample sizes ($n \leq 30$). For a sample size of 20 and $\alpha = 0.01$, $r_{0.01} = 0.534$ for a one-tailed test.

5.3.4 Results

5.3.4.1 Data (i.e. raw data collected during study)

Each of the JDK versions (viz. JDK1.1.7 through J2SDK1.4.0) and ANT versions (ANT1.1. through ANT1.5) were given to the metrics SSSM, SBSM, and RBSM for self-comparison. In all cases, the similarity metrics produced a maximal value of unity. This strongly supports hypotheses $H_{1,**SM}$. The raw values have not been shown here because of their repetitive nature. A small subset of them can however be seen in Table 5, 8, 10, 13, 15, and 18.

Every pair of models compared and shown in Table 5, 8, 10, 13, 15, and 18 are compared in reverse order. In other words, for every pair of models, whenever we compute a similarity metric for m_1 and m_2 , we also compute it for m_2 and m_1 . In all cases, we found the pair of computed values to be identical. This strongly supports hypotheses $H_{2,**SM}$.

Table 5: Comparing JDK and J2SDK Versions using SSSM

Model	Number of classes	JDK1.1.7				J2SDK1.4.0			
		Intuitive Rank	SSSM		SSSM Rank	Intuitive Rank	SSSM		SSSM Rank
			Greedy	1-1 Heuristic			Greedy	1-1 Heuristic	
JDK1.1.7	84	1	1.0	1.0	1	5	0.610	0.603	5
JDK1.2.2	115	2	0.727	0.718	2	4	0.836	0.832	4
J2SDK1.3.0	122	3	0.685	0.677	3	3	0.887	0.883	2
J2SDK1.3.1	122	4	0.685	0.677	3	2	0.887	0.883	2
J2SDK1.4.0	137	5	0.610	0.603	5	1	1.0	1.0	1

JDK/J2SDK Comparisons with SSSM

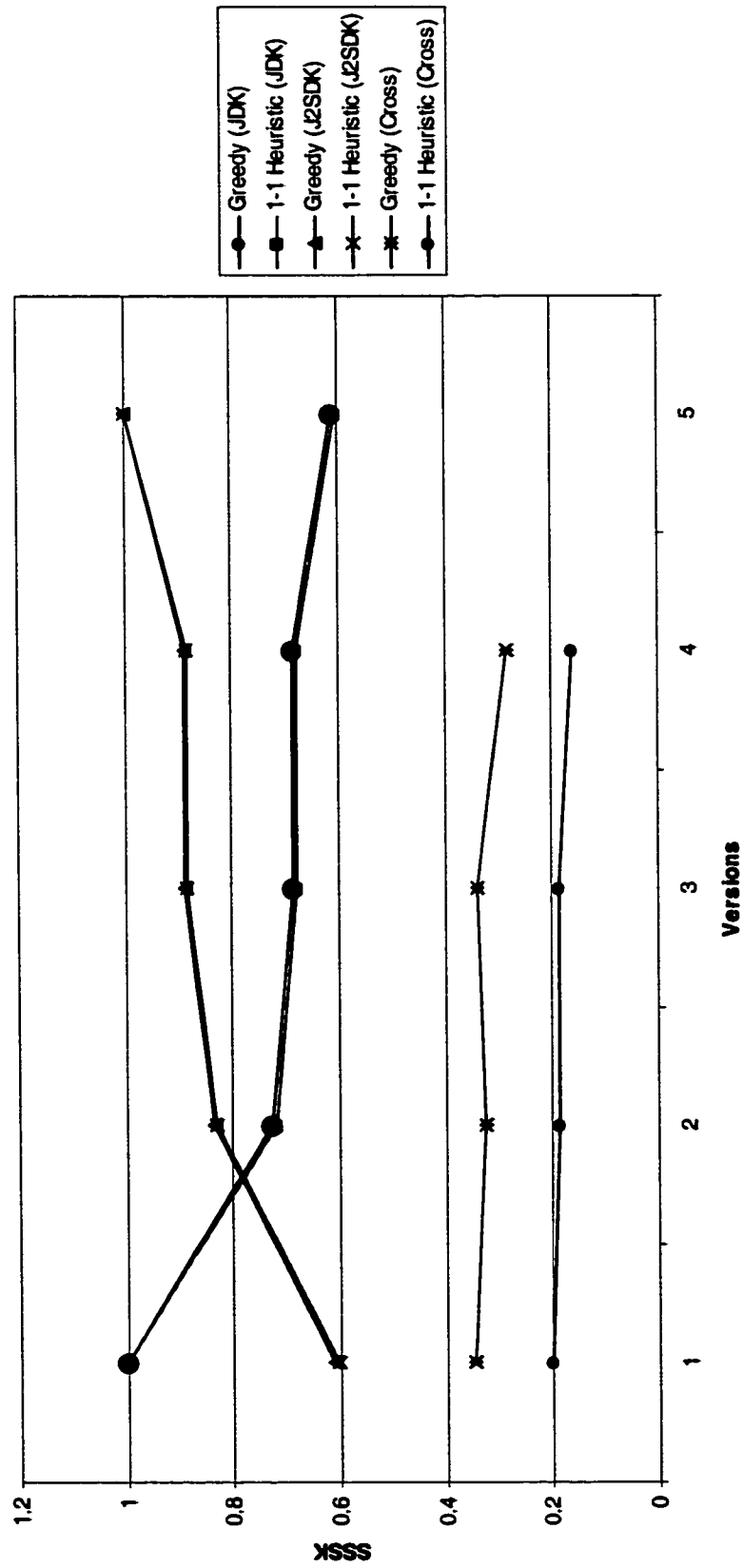


Figure 6: Comparing JDK versions using SSSM

An observation of Figure 6 above shows that when we compare the JDK versions with JDK 1.1.7 using SSSM, we see a gradual decrease in similarity values. When JDK 1.1.7 is compared with itself, we obtain the maximum value of 1.0. The values drop gradually as we compare JDK 1.1.7 with later versions, until we obtain a value of about 0.6 when it is compared with J2SDK 1.4.0. This indicates that as additional parts are added to one of the models, SSSM similarity values decrease gradually. This supports the Intuition that similarity reduces with growing differences (i.e. Intuition 3). This observation holds true for both the Greedy version as well as the 1-1 Heuristic version of our metric, although the values obtained from the latter tend to be slightly less in nearly all cases investigated.

A similar observation can be made from Figure 7 below concerning Apache ANT versions. A comparison of ANT 1.1 with itself gives a maximal SSSM value of 1.0. Comparing with successive versions of the ANT, we obtain decreasing SSSM values, with one exception: ANT 1.3 reports an increase rather than a decrease. This becomes understandable when we note that the number of classes in ANT 1.3 reduced from 147 in ANT 1.2 to 146, suggesting that ANT 1.3 may just have been a reorganization or a refactoring of ANT 1.2 classes.

On comparing the JDK versions against J2SDK 1.4.0, we see a gradual increase in the SSSM values. When J2SDK 1.4.0 is compared with JDK1.1.7 we obtain an SSSM value in the neighborhood of 0.6 as we can see from Figure 6. The SSSM value grows gradually as we compare J2SDK 1.4.0 with later versions of the JDK (i.e. 1.2.2 through 1.4.0), and reaches a maximum value of 1.0 when J2SDK 1.4.0 is compared with itself. This observation supports the intuition that similarity increases with commonality (Intuition 3).

In the same fashion, Figure 7 also includes a comparison of ANT 1.5 against the other versions of Apache ANT. A gradual growth in the values of the SSSM similarity metric is observed as expected. This also supports Intuition 3 as discussed above.

It is interesting to note that the *proportionate change*, Δ (see Equation 23 below), in the SSSM values from one comparison to the next somewhat correlates with the proportionate change in the number of classes in the corresponding models from the previous version. We define a proportionate change, Δ , between the values a_1 and a_2 as follows:

$$\Delta = \frac{|a_1 - a_2|}{\min(a_1, a_2)} \quad 23$$

Table 6 and 7 below show this correlation for the JDK and the ANT versions respectively.

Table 6: Correlation between proportionate changes in number of classes and SSSM values in the JDK versions

Models	# of classes	Δ in # of classes	JDK1.1.7		J2SDK1.4.0	
			Greedy Δ	1-1 Heuristic Δ	Greedy Δ	1-1 Heuristic Δ
JDK 1.1.7	84	-	-	-	-	-
JDK 1.2.2	115	0.37	0.38	0.39	0.37	0.38
J2SDK1.3.0	122	0.06	0.06	0.06	0.06	0.06
J2SDK1.3.1	122	0.00	0.00	0.00	0.00	0.00
J2SDK1.4.0	137	0.12	0.12	0.12	0.13	0.13

Table 7: Correlation between proportionate changes in number of classes and SSSM values in the ANT versions

Models	# of classes	Δ in # of classes	ANT1.1		ANT1.5	
			Greedy Δ	1-1 Heuristic Δ	Greedy Δ	1-1 Heuristic Δ
ANT 1.1	97	-	-	-	-	-
ANT 1.2	147	0.52	0.53	0.54	0.37	0.36
ANT 1.3	146	0.01	0.01	0.01	0.11	0.12
ANT 1.4	204	0.40	0.40	0.40	0.38	0.38
ANT 1.5	318	0.56	0.58	0.58	0.61	0.62

Figure 6 and Figure 7 also show the result of comparing JDK versions against ANT versions (labeled “cross” in the legends of Figure 6 and Figure 7). As one would expect, the SSSM values of these comparisons are much lower than the values obtained when JDK versions are compared with one another or when ANT versions are compared among themselves.

Table 8: Comparing Apache ANT Versions using SSSM

Model	Number of classes	Apache ANT 1.1				Apache ANT 1.5			
		Intuitive Rank	SSSM		SSSM Rank	Intuitive Rank	SSSM		SSSM Rank
			Greedy	1-1 Heuristic			Greedy	1-1 Heuristic	
Apache ANT 1.1	97	1	1.0	1.0	1	5	0.298	0.295	5
Apache ANT 1.2	147	2	0.652	0.649	3	4	0.408	0.400	4
Apache ANT 1.3	146	3	0.657	0.653	2	3	0.452	0.449	3
Apache ANT 1.4	204	4	0.470	0.465	4	2	0.622	0.618	2
Apache ANT 1.5	318	5	0.298	0.295		1	1.0	1.0	1

ANT/ANT Comparison using SSSM

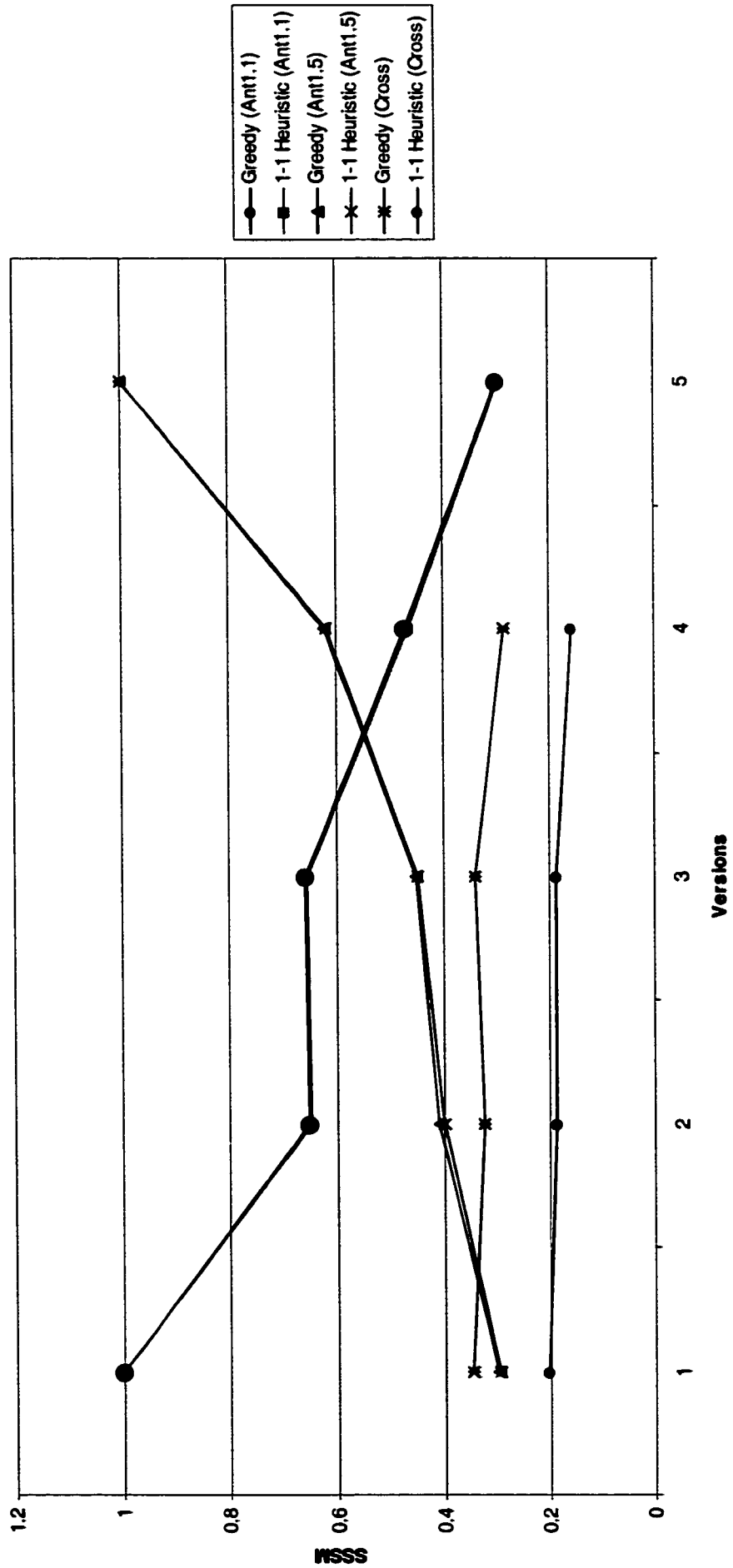


Figure 7: Comparing ANT versions using SSSM

The table below shows the computation of the spearman's rank correlation of the SSSM data

Table 9: Rank Correlation Computation for SSSM

	Intuitive Ranks	SSSM Ranks	D	D^2
1	1	1	0	0
2	2	2	0	0
3	3	3	0	0
4	4	3	1	1
5	5	5	0	0
6	5	5	0	0
7	4	4	0	0
8	3	2	1	1
9	2	2	0	0
10	1	1	0	0
11	1	1	0	0
12	2	3	-1	1
13	3	2	1	1
14	4	4	0	0
15	5	5	0	0
16	5	5	0	0
17	4	4	0	0
18	3	3	0	0
19	2	2	0	0
20	1	1	0	0
	ΣD^2			4
	N			20
	Spearman's rank correlation Coefficient, r_s .			0.996992

Since $r_s = 0.996992$ does not fall in the rejection region (i.e. $r_s \nless r_{0.01} = 0.534$), we conclude that there is a direct positive correlation between intuitive rankings and SSSM with 99% confidence.

Table 10: Comparing JDK and J2SDK Versions using SBSM

Model	# of classes	JDK1.1.7				J2SDK1.4.0			
		Intuitive Rank	SBSM		SBSM Rank	Intuitive Rank	SBSM		SBSM Rank
			Greedy	1-1 Heuristic			Greedy	1-1 Heuristic	
JDK1.1.7	84	1	1.0	1.0	1	5	0.5642	0.5484	5
JDK1.2.2	115	2	0.6828	0.6538	2	4	0.8057	0.7748	4
J2SDK1.3.0	122	3	0.6454	0.6347	3	3	0.8587	0.8255	2
J2SDK1.3.1	122	4	0.6454	0.6347	3	2	0.8587	0.8255	2
J2SDK1.4.0	137	5	0.5642	0.5484	5	1	1.0	1.0	1

JDK/J2SDK Comparisons with SBSM

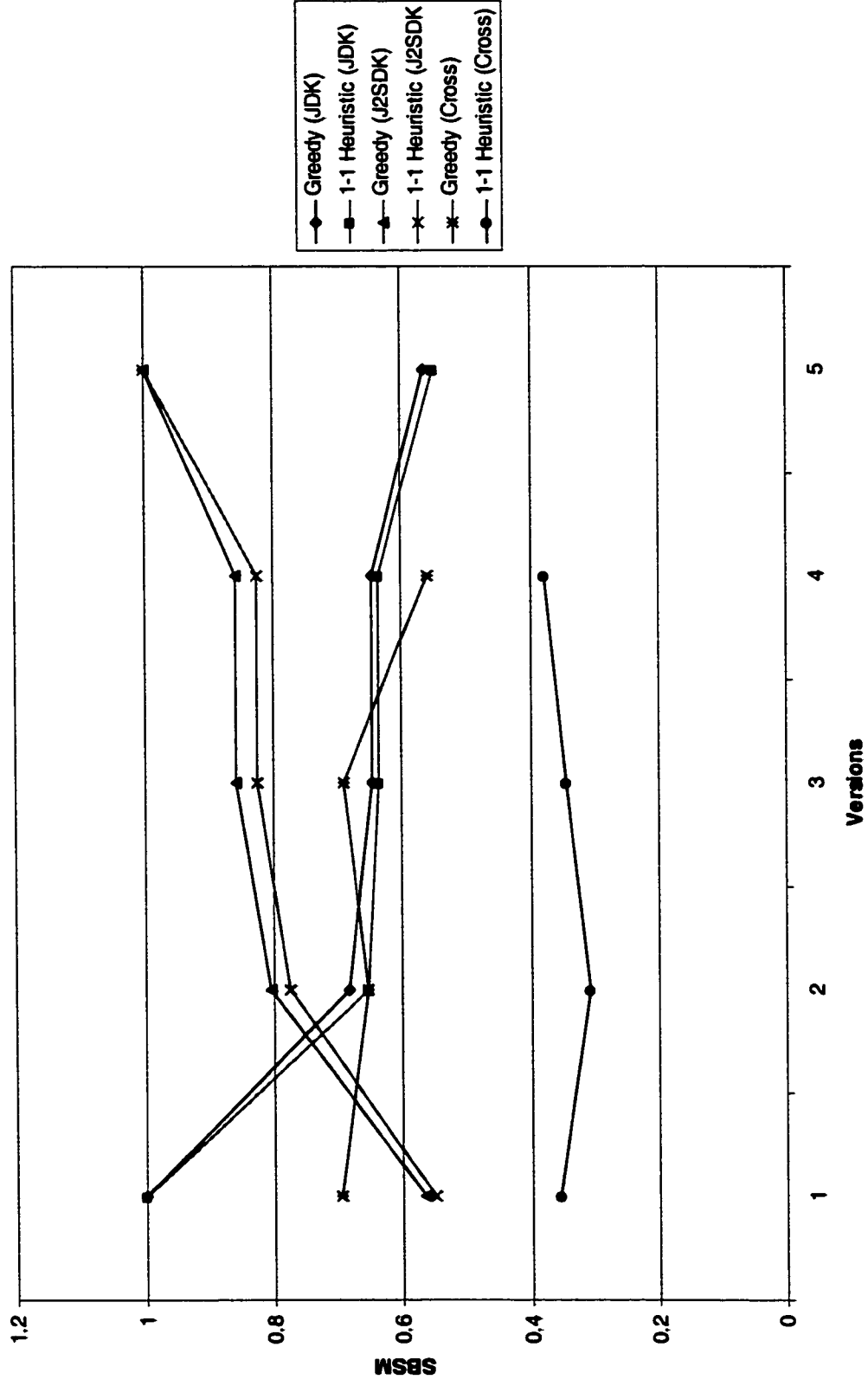


Figure 8: Comparing JDK versions using SBSM

An observation of Figure 8 above shows that when we compare the JDK versions with JDK 1.1.7 using SBSM, we see a gradual decrease in similarity values. When JDK 1.1.7 is compared with itself, we obtain the maximum value of 1.0. The values drop gradually as we compare JDK 1.1.7 with later versions, until we obtain a value of about 0.5 when it is compared with J2SDK 1.4.0. This indicates that as additional parts are added to one of the models, SBSM similarity values decrease gradually. This supports the Intuition that similarity reduces with growing differences (i.e. Intuition 3). This observation holds true for both the Greedy version as well as the 1-1 Heuristic version of our metric, although the values obtained from the latter tend to be slightly less in nearly all cases investigated.

A similar observation can be made from Figure 9 below concerning Apache ANT versions. A comparison of ANT 1.1 with itself gives a maximal SBSM value of 1.0. Comparing with successive versions of the ANT, we obtain decreasing SBSM values, with one exception: ANT 1.3 reports an increase rather than a decrease in the case of the 1-1 heuristic version of SBSM. This is explained by the fact that the number of classes in ANT 1.3 reduced from 147 in ANT 1.2 to 146, suggesting that ANT 1.3 may just have been a reorganization or a refactoring of ANT 1.2 classes. In that case, the reliability of the Greedy version of SBSM is then open to question.

On comparing the JDK versions against J2SDK 1.4.0, we see a gradual increase in the SBSM values. When J2SDK 1.4.0 is compared with JDK1.1.7 we obtain an SBSM value in the neighborhood of 0.5 as we can see from Figure 8. The SBSM value grows gradually as we compare J2SDK 1.4.0 with later versions of the JDK (i.e. 1.2.2 through 1.4.0), and reaches a climax when J2SDK 1.4.0 is compared with itself. This observation supports the intuition that similarity increases with commonality (Intuition 3).

In the same fashion, Figure 9 also includes a comparison of ANT 1.5 against the other versions of Apache ANT. A gradual growth in the values of the SBSM similarity metric is observed as expected. This also supports Intuition 3 as discussed above.

It is interesting to note that the *proportionate change*, Δ (see Equation 23 above), in the SBSM values from one comparison to the next somewhat correlates with the proportionate change in the number of classes in the corresponding models from the previous version.

Table 11 and Table 12 below show this correlation for the JDK and the ANT versions respectively.

Table 11: Correlation between proportionate changes in # of classes and SBSM values in the JDK versions

Models	# of classes	Δ in # of classes	JDK1.1.7		J2SDK1.4.0	
			Greedy Δ	1-1 Heuristic Δ	Greedy Δ	1-1 Heuristic Δ
JDK 1.1.7	84	-	-	-	-	-
JDK 1.2.2	115	0.37	0.46	0.53	0.43	0.41
J2SDK1.3.0	122	0.06	0.06	0.03	0.07	0.07
J2SDK1.3.1	122	0.00	0.00	0.00	0.00	0.00
J2SDK1.4.0	137	0.12	0.14	0.16	0.16	0.21

Table 12: Correlation between proportionate changes in # of classes and SBSM values in the ANT versions

Models	# of classes	Δ in # of classes	ANT1.1		ANT1.5	
			Greedy Δ	1-1 Heuristic Δ	Greedy Δ	1-1 Heuristic Δ
ANT 1.1	97	-	-	-	-	-
ANT 1.2	147	0.52	0.71	0.85	0.50	0.41
ANT 1.3	146	0.01	0.01	0.01	0.05	0.14
ANT 1.4	204	0.40	0.41	0.46	0.45	0.47
ANT 1.5	318	0.56	0.57	0.54	0.68	0.75

Both Figure 8 and Figure 9 also show the result of comparing JDK versions against ANT versions (labeled “cross” in the legend of Figure 8 and Figure 9). One would have expected the SBSM values of these comparisons to be lower than are those that would be obtained when we compare versions of the same product, since versions of the same product are supposed to have more things in common and fewer things in difference. However, what we observe is that this is more or less true for the 1-1 Heuristic version of SBSM. The Greedy version is producing results that conflict with the intuition that models having greater commonality (in this case, versions of

the same product) have greater similarity than models having fewer things in common (in this case models of different products). This leads us to believe that the 1-1 Heuristic version is more reliable than its Greedy counterpart is.

Table 13: Comparing Apache ANT Versions using SBSM

Model	# of classes	Apache ANT 1.1				Apache ANT 1.5			
		Intuitive Rank	SBSM		SBSM Rank	Intuitive Rank	SBSM		SBSM Rank
			Greedy	1-1 Heuristic			Greedy	1-1 Heuristic	
Apache ANT 1.1	97	1	1.0	1.0	1	5	0.2593	0.2426	5
Apache ANT 1.2	147	2	0.5836	0.5414	2	4	0.3891	0.3410	4
Apache ANT 1.3	146	3	0.5765	0.5456	3	3	0.4096	0.3876	3
Apache ANT 1.4	204	4	0.4083	0.3737	4	2	0.5939	0.5707	2
Apache ANT 1.5	318	5	0.2593	0.2426	5	1	1.0	1.0	1

ANT/ANT Comparison using SBSM

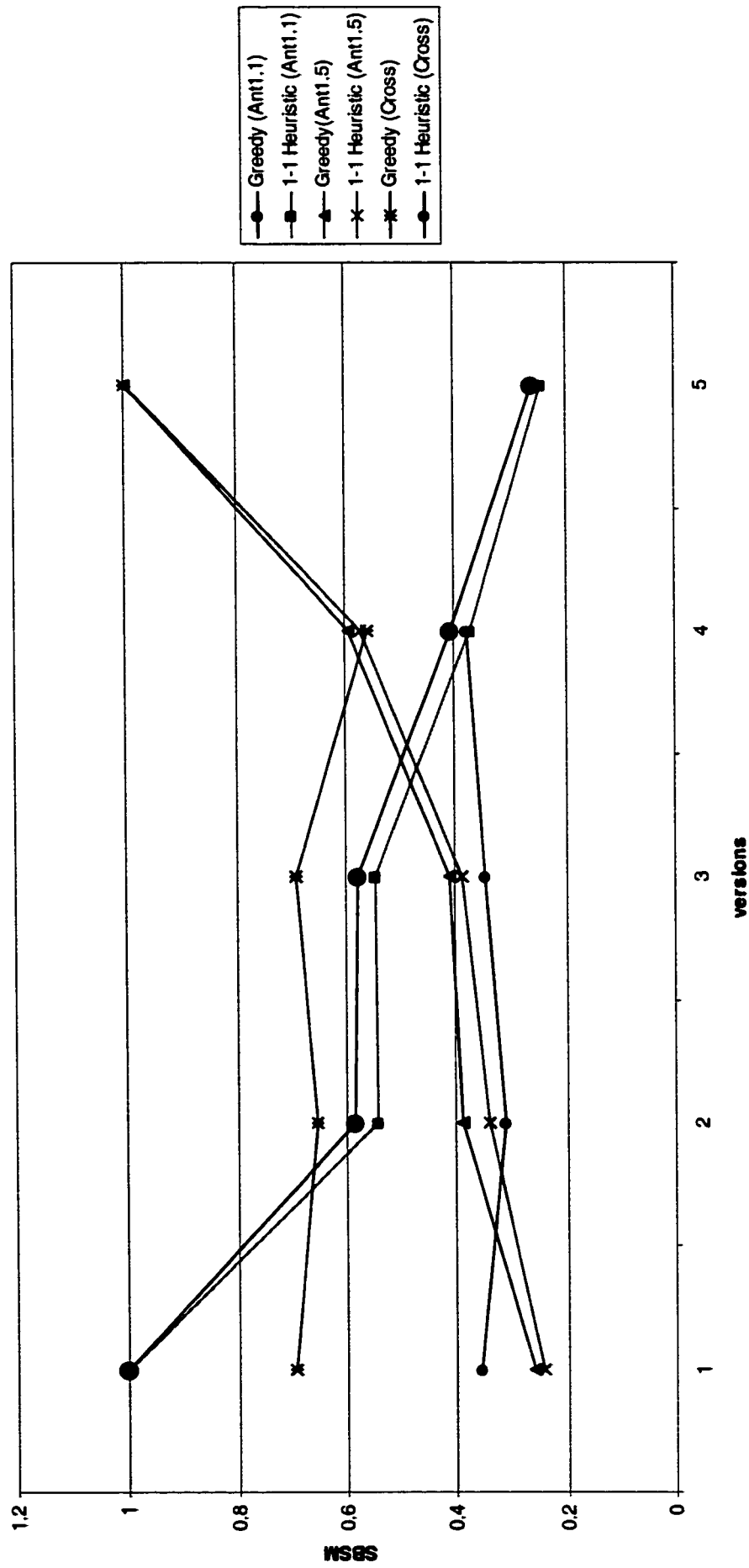


Figure 9: Comparing ANT versions using SBSM

Table 14 below shows the computation of the Spearman's rank correlation coefficient for the SBSM values.

Table 14: Rank Correlation Computation for SBSM

	Intuitive Ranks	SBSM Ranks	D	D^2
1	1	1	0	0
2	2	2	0	0
3	3	3	0	0
4	4	3	1	1
5	5	5	0	0
6	5	5	0	0
7	4	4	0	0
8	3	2	1	1
9	2	2	0	0
10	1	1	0	0
11	1	1	0	0
12	2	2	0	0
13	3	3	0	0
14	4	4	0	0
15	5	5	0	0
16	5	5	0	0
17	4	4	0	0
18	3	3	0	0
19	2	2	0	0
20	1	1	0	0
	ΣD^2			2
	N			20
	Spearman's rank correlation Coefficient, r_s .			0.998496241

Since $r_s = 0.998496241$ does not fall in the rejection region (i.e. $r_s \not\leq r_{0.01} = 0.534$), we conclude that there is a direct positive correlation between intuitive rankings and SBSM with 99% confidence.

Table 15: Comparing JDK and J2SDK Versions using RBSM

Model	# of classes	JDK1.1.7					J2SDK1.4.0			
		Intuitive Rank	RBSM		RBSM Rank	Intuitive Rank	RBSM		RBSM Rank	
			Greedy	1-1 Heuristic			Greedy	1-1 Heuristic		
JDK1.1.7	84	1	1.0	1.0	1	5	0.613	0.613	5	
JDK1.2.2	115	2	0.730	0.730	2	4	0.839	0.839	4	
J2SDK1.3.0	122	3	0.688	0.688	3	3	0.890	0.890	2	
J2SDK1.3.1	122	4	0.688	0.688	3	2	0.890	0.890	2	
J2SDK1.4.0	137	5	0.613	0.613	5	1	1.0	1.0	1	

JDK/J2SDK Comparison with RBSM

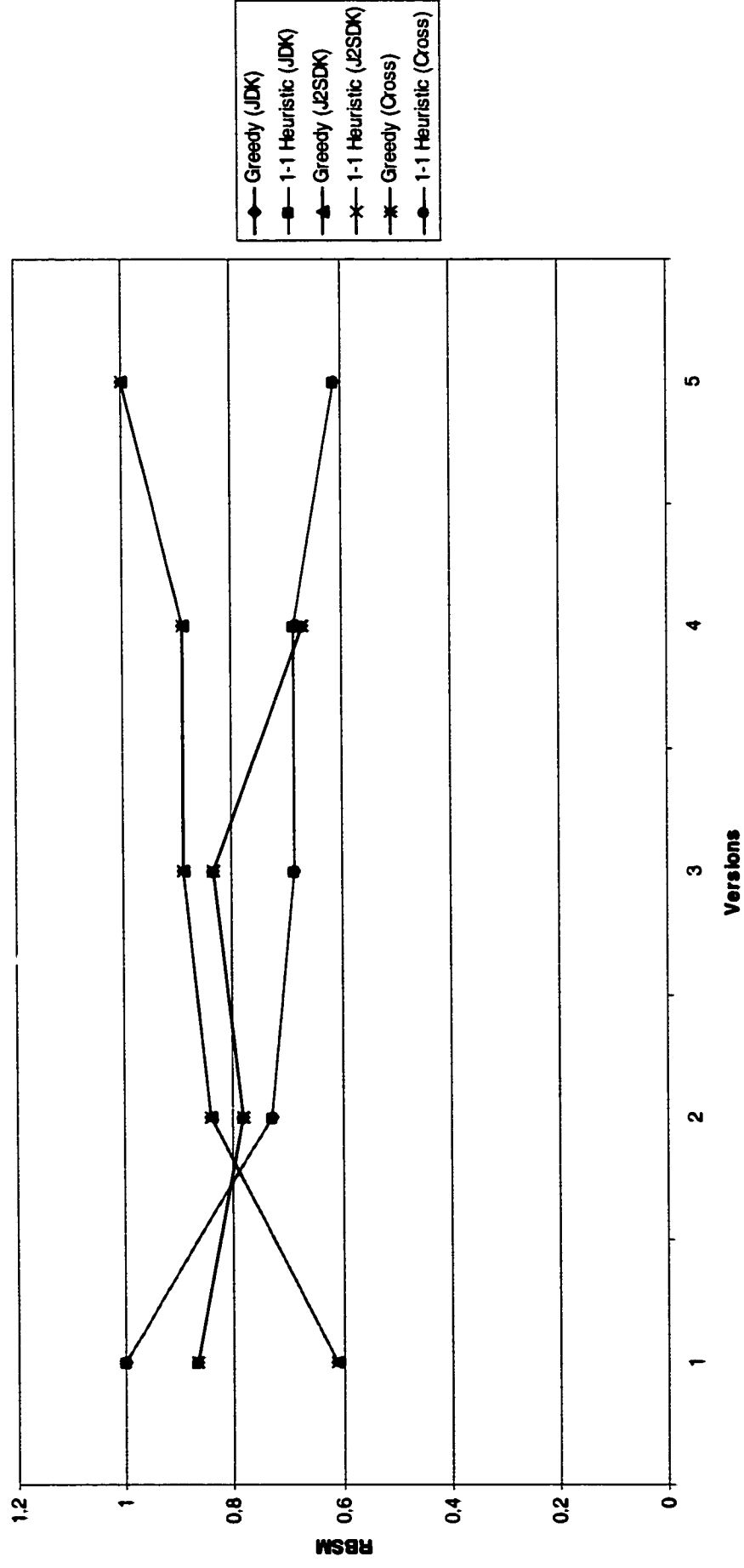


Figure 10: Comparing JDK versions using RBSM

The first thing to notice from Figure 10 and Figure 11 is that both the Greedy and the 1-1 Heuristic versions of RBSM produce identical results. This suggests that the additional overhead involved in computing the 1-1 Heuristic version is unnecessary. An observation of Figure 10 above shows that when we compare the JDK versions with JDK 1.1.7 using RBSM, we see a gradual decrease in similarity values. When JDK 1.1.7 is compared with itself, we obtain the maximum value of 1.0. The values drop gradually as we compare JDK 1.1.7 with later versions, until we obtain a value of about 0.6 when it is compared with J2SDK 1.4.0. This indicates that as additional parts are added to one of the models, RBSM similarity values decrease gradually. This supports the Intuition that similarity reduces with growing differences (i.e. Intuition 3).

A similar observation can be made from Figure 11 below concerning Apache ANT versions. A comparison of ANT 1.1 with itself gives a maximal RBSM value of 1.0. Comparing with successive versions of the ANT, we obtain decreasing RBSM values, with one exception: ANT 1.3 reports an increase rather than a decrease. This becomes understandable when we note that the number of classes in ANT 1.3 reduced from 147 in ANT 1.2 to 146, suggesting that ANT 1.3 may just have been a reorganization or a refactoring of ANT 1.2 classes.

On comparing the JDK versions against J2SDK 1.4.0, we see a gradual increase in the RBSM values. When J2SDK 1.4.0 is compared with JDK1.1.7 we obtain an RBSM value in the neighborhood of 0.6 as we can see from Figure 10. The RBSM value grows gradually as we compare J2SDK 1.4.0 with later versions of the JDK (i.e. 1.2.2 through 1.4.0), and reaches a maximum value of 1.0 when J2SDK 1.4.0 is compared with itself. This observation supports the intuition that similarity increases with commonality (Intuition 3).

In the same fashion, Figure 11 also includes a comparison of ANT 1.5 against the other versions of Apache ANT. A gradual growth in the values of the RBSM similarity metric is observed as expected. This also supports Intuition 3 as discussed above.

It is interesting to note that the *proportionate change*, Δ (see Equation 23 below), in the RBSM values from one comparison to the next somewhat correlates with the proportionate change in the number of classes in the corresponding models from the previous version.

Table 16 and Table 17 below show this correlation for the JDK and the ANT versions respectively.

Table 16: Correlation between proportionate changes in number of classes and RBSM values in the JDK versions

Models	# of classes	Δ in # of classes	JDK1.1.7		J2SDK1.4.0	
			Greedy Δ	1-1 Heuristic Δ	Greedy Δ	1-1 Heuristic Δ
JDK 1.1.7	84	-	-	-	-	-
JDK 1.2.2	115	0.37	0.37	0.37	0.37	0.37
J2SDK1.3.0	122	0.06	0.06	0.06	0.06	0.06
J2SDK1.3.1	122	0.00	0.00	0.00	0.00	0.00
J2SDK1.4.0	137	0.12	0.12	0.12	0.12	0.12

Table 17: Correlation between proportionate changes in number of classes and RBSM values in the ANT versions

Models	# of classes	Δ in # of classes	ANT1.1		ANT1.5	
			Greedy Δ	1-1 Heuristic Δ	Greedy Δ	1-1 Heuristic Δ
ANT 1.1	97	-	-	-	-	-
ANT 1.2	147	0.52	0.52	0.52	0.51	0.51
ANT 1.3	146	0.01	0.01	0.01	0.01	0.01
ANT 1.4	204	0.40	0.40	0.40	0.40	0.40
ANT 1.5	318	0.56	0.56	0.56	0.56	0.56

Figure 10 and Figure 11 also show the result of comparing JDK versions against ANT versions (labeled “cross” in the legends of Figure 10 and Figure 11). Contrary to expectation, the RBSM values of these comparisons are not any lower than the values obtained when JDK versions (or ANT versions) are compared with one another. This conflicts with Intuition 3, which states that models that have greater commonality (in this case versions of same product) should have greater similarity than models that have fewer things in common (in this case models of different products). This is an indication that RBSM is not so reliable in this case. The reason for this is that relationships can sometimes be nonexistent in a class, and at other times, several classes

can derive from a common superclass, such as the Object class in Java-based systems. When such classes pervade the two models being compared, one would often end up with relationship sets that are identical in both models. This will then lead to the spuriously high similarity values obtained from by RBSM. One solution to this problem is to combine RBSM with one of the other similarity metrics discussed earlier, say in a cascade fashion. However, in some other cases such as when models are tightly coupled or when models being compared are within the same product line, RBSM produces reliable results.

Table 18: Comparing Apache ANT Versions using RBSM

Model	# of classes	Apache ANT 1.1				Apache ANT 1.5			
		Intuitive Rank	RBSM		RBSM Rank	Intuitive Rank	RBSM		RBSM Rank
			Greedy	1-1 Heuristic			Greedy	1-1 Heuristic	
Apache ANT 1.1	97	1	1.0	1.0	1	5	0.305	0.305	5
Apache ANT 1.2	147	2	0.659	0.659	3	4	0.462	0.462	3
Apache ANT 1.3	146	3	0.664	0.664	2	3	0.459	0.459	4
Apache ANT 1.4	204	4	0.475	0.475	4	2	0.641	0.641	2
Apache ANT 1.5	318	5	0.305	0.305	5	1	1.0	1.0	1

ANT/ANT Comparison with RBSM

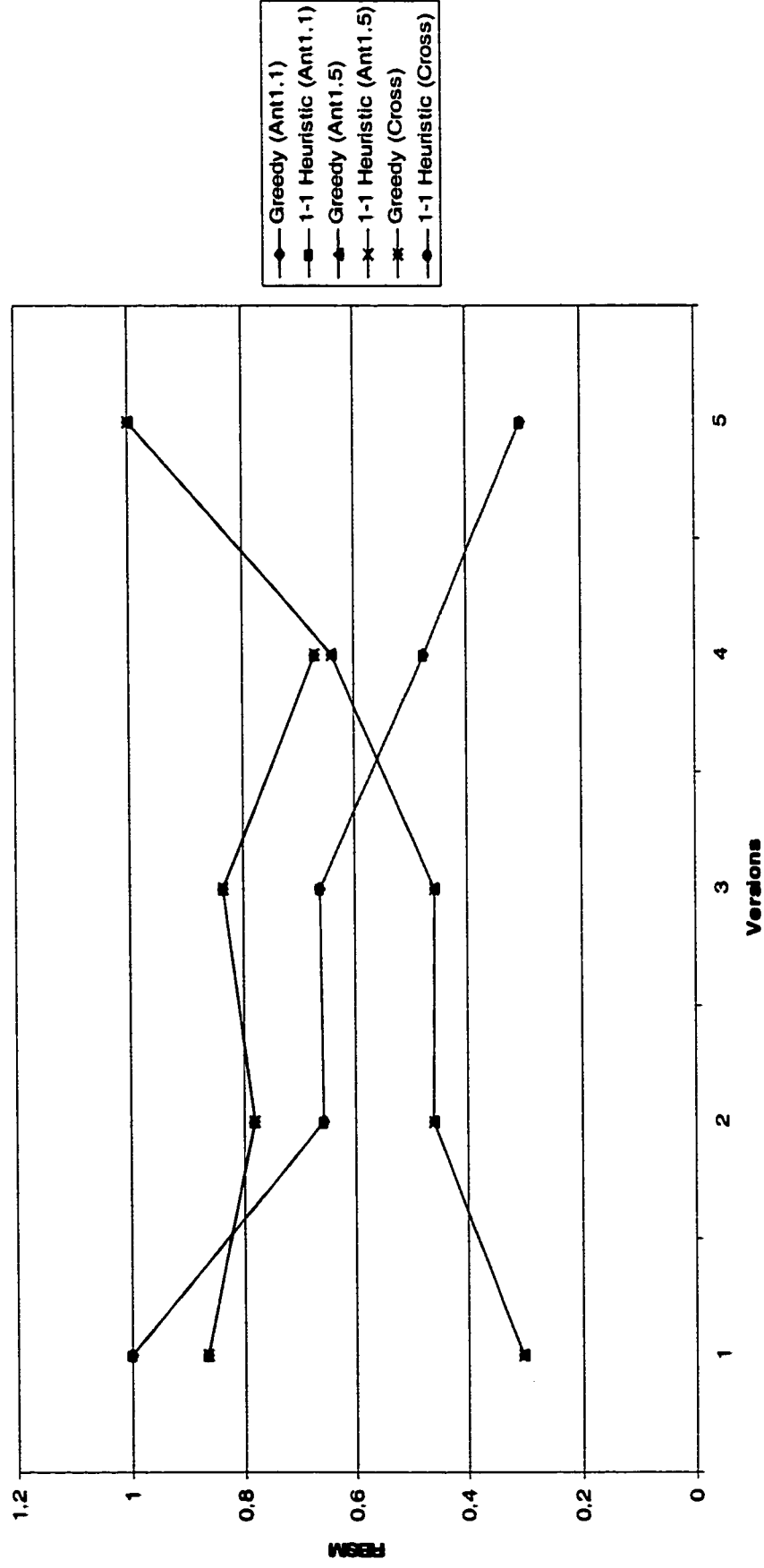


Figure 11: Comparing ANT versions using RBSM

Table 19 below shows the computation of the Spearman's rank correlation coefficient for the RBSM values.

Table 19: Rank Correlation Computation for RBSM

	Intuitive Ranks	RBSM Ranks	D	D^2
1	1	1	0	0
2	2	2	0	0
3	3	3	0	0
4	4	3	1	1
5	5	5	0	0
6	5	5	0	0
7	4	4	0	0
8	3	2	1	1
9	2	2	0	0
10	1	1	0	0
11	1	1	0	0
12	2	3	-1	1
13	3	2	1	1
14	4	4	0	0
15	5	5	0	0
16	5	5	0	0
17	4	3	1	1
18	3	4	-1	1
19	2	2	0	0
20	1	1	0	0
	ΣD^2			6
	N			20
	Spearman's Rank Correlation Coefficient, r_s.			0.995488722

Since $r_s = 0.995488722$ does not fall in the rejection region (i.e. $r_s \not\leq r_{0.01} = 0.534$), we conclude that there is a direct positive correlation between intuitive rankings and RBSM with 99% confidence.

Table 20: Comparing JDK against Apache ANT

	SSSM		SBSM		RBSM	
	Greedy	1-1 Heuristic	Greedy	1-1 Heuristic	Greedy	1-1 Heuristic
JDK1.1.7 vs. ANT1.1	0.3475	0.2011	0.6963	0.3568	0.865	0.865
JDK1.2.2 vs. ANT1.2	0.3250	0.1860	0.6538	0.3108	0.782	0.782
J2SDK1.3.0 vs. ANT1.3	0.3385	0.1851	0.6922	0.3467	0.835	0.835
J2SDK1.4.0 vs. ANT 1.4.1	0.2842	0.1584	0.5594	0.3778	0.671	0.671

Comparing these results to those obtained when comparing versions of the same product, one would expect that these values would be lower than values obtained from comparing versions of the same product. However, this has not been the case for SBSM and RBSM. This can be explained by the fact that class signatures and fingerprints are not very distinctive. It is easily possible to have identical signature and fingerprints from classes belonging to completely different products. The same explanation goes for SBSM, since attribute types and signatures are not very distinctive either, especially with the Greedy version.

Observing the SSSM columns of Table 20, however, we find the values to be generally lower than the values observed for comparisons within the same

product lines. This suggests that SSSM is more sensitive to differences than SBSM and RBSM. Since our previous observations have suggested that the three metrics are comparative in their similarity assessment, for comparisons of versions of the same product (*cf.* Table 5 through Table 19), the current observation suggests that the three metrics rank as follows in terms of overall agreement with human intuition (from best to worst): SSSM, SBSM, and RBSM. SSSM appears to be the most reliable. It is followed by SBSM and RBSM appears to be the least reliable of the three. By reliability here, we mean the ability to assess similarity accurately irrespective of whether models under comparison come from the same domain or not.

5.3.4.2 Interpretations (i.e. statements about the hypotheses)

The rankings based on shallow semantic similarity metric (SSSM), signature-based similarity metric (SBSM), and relationship-based similarity metric (RBSM) show near perfect correlation with our intuitive rankings. The Deep semantic similarity metric (DSSM) could not be executed to completion on the treatments, due to its excessive time complexity.

The tendency of developers to abbreviate identifier names, not to always capitalize acronyms and beginnings of words within an identifier, can somewhat reduce the effectiveness of semantic similarity metrics in general. Table 21 below shows the distribution of these kinds of naming anomalies

found in the ANT and JDK packages studied. Abbreviated words and multiple words without initial capitals account for up to about 12% and 11% of identifiers in ANT 1.3 for instance. This can greatly influence the effectiveness of semantically based similarity metrics, in general.

Table 21: Distribution of Naming Anomalies

	Number of classes	Uncapitalized Acronyms		Shortened Words		Multiple words without initial capitals	
		#	%	#	%	#	%
JDK1.1.7	84	0	0.00%	0	0.00%	1	1.19%
JDK1.2.2	115	0	0.00%	0	0.00%	1	0.87%
J2SDK1.3.0	122	0	0.00%	0	0.00%	1	0.82%
J2SDK1.4.0	137	0	0.00%	0	0.00%	1	0.73%
ANT1.1	97	3	3.09%	7	7.22%	7	7.22%
ANT1.2	147	5	3.40%	11	7.48%	15	10.20%
ANT1.3	146	5	3.42%	17	11.64%	16	10.96%
ANT1.4.1	204	10	4.90%	14	6.86%	14	6.86%
ANT1.5	318	21	6.60%	25	7.86%	10	3.14%
Ave.	152.22	4.89	2.38%	8.22	4.56%	7.33	4.67%

An inspection of the classes studied shows that class names across product versions tend not to change at all. This means that within a product line, exact matches tend to be frequent.

Chapter 6

Conclusion

6.1 Introduction

In this chapter, we give a summary of our work and give indications of how it can be improved upon in the future.

6.2 Summary and Contributions of Thesis

In this thesis, we have presented a survey of the literature of software similarity assessment approaches and touched upon the attendant issues of software artifact retrieval, artifact representation, repository technology and the software product lines approach.

We have described conceptually an approach for matching UML models in a manner that accounts for the multiple views of UML models.

We have devised, implemented, and validated a suite of metrics for matching UML class models.

The empirical results reported indicate high positive correlation with human intuitive similarity judgments.

6.3 Limitations and Further Work

Further work is required to devise and validate similarity metrics for the other views (and diagrams) of the UML, such as the dynamic and use-case views (sequence diagrams and use case diagrams).

In Section 3.3, we suggested that the difference between two models would not be used in our similarity assessment. A future work may further investigate this assumption by defining a difference metric (i.e. a *diff* function) for each of our proposed metrics, incorporate that into the metric, and compare results that will be obtained with ours.

In Section 3.5, we suggested a formula for the computing the inconsistency penalty, I , that indicates the degree of inconsistency among a set of metrics that are composed together hierarchally. Further is needed to generalize this formula and to validate its use.

In Section 4.2.1, we mentioned that there are other semantic distance measures besides HSO. Further work is needed to compare results of using other semantic distance measures in our metrics with current results.

We also stated in Section 4.2.1 that our current implementation makes use of a Perl script [80] developed by Ted Pederson and his team in the University

of Minnesota. Interfacing Java with Perl has performance penalties. Further work is needed in porting these metrics to Java for better performance.

Further work is also required to implement and experiment with the cascade metaphor discussed in Section 3.4. The cascade metaphor allows metrics within the same view to be composed together in a pipeline fashion. Several different cascades may be implemented and their results compared with each other and to our present results.

In Sections 4.2.3, 4.2.4, and 4.2.5, we stated that our current implementation uses fixed weights in the computation of our metric set. A dynamic weight determination scheme or at least a more rigorous weight determination approach can be pursued a future work.

In the implementation of DSSM and SSSM, we send each pair of identifier names (after tokenizing and shredding off numbers, etc.), to the semantic distance server to compute the semantic distance for the pair. Sometimes, pairs could be sent multiple times if they occur in multiple parts of the models. Further work is required to improve on this by first generating a *set* of pairs for which semantic distances will be calculated in a batch. The set is then sent to the server all at once rather than each pair at a time. This way, performance will be boosted considerably.

In Section 4.2.3.1 and Table 21, we identified a number of naming anomalies that are often found in models and source codes. Further work will be required to implement algorithms that can detect and cope with such anomalies. For instance, disambiguating short words and determining the actual word they stand for can probably benefit from work in word sense disambiguation [15][97]; and tokenizing words with uncapitalized initial capitals can benefit from research results in spelling correction [53].

In Section 5.3.3.3, we stated that our current implementation accepts UML models in XMI 1.0 for UML 1.3 as input. Further work is required so that our tool can accommodate a wider range of input formats, especially the newer versions of the XMI format.

In Section 4.2.4, we stated that our current implementation of SBSM, the `typesim()` routine simply does an exact string match when comparing attribute types and method signatures. It does not take into consideration scope subsumption, specialization, generalization, and interface realization relationships that exist among types. Determining type equivalency in SBSM can take advantage of classification trees in a future work for greater sensitivity to these considerations.

Bibliography

- [1] Aiken, A. MOSS (Measure of Software Similarity) Plagiarism Detection System. (<http://www.cs.berkeley.edu/~moss/>). University of Berkeley, CA.
- [2] Alsuwaiyel, M.H. *Algorithms: Design Techniques and Analysis*. World Scientific, 1999.
- [3] Antoniol, G., A. Potrich, P. Tonella and R. Fiutem, "Evolving Object Oriented Design to Improve Code Traceability". *Proc. of the International Workshop on Program Comprehension (IWPC)*, pp. 151-160, Pittsburgh, PA, USA, May 5-7, 1999.
- [4] Atkinson, S. *Formal Engineering of Software Library Systems*. PhD Thesis, Dept. of Comp. Sc. & Elec. Eng., Univ. of Queensland, 1997.
- [5] Basili, V., F. Shull, and F. Lanubile, Using Experiments to Build a Body of Knowledge, *Proceedings of the Third International PSI Conference*, Novosibirsk, Russia, pp. 265-282, July 1999.
- [6] Basili, V.R., G. Caldiera, and H.D. Rombach. Goal Question Metric Paradigm. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, pages 528-532. Wiley & Sons, 1994
- [7] Bayer, J., J.-F Girard, M. Wuerthner, J.-M. DeBaud, and M. Apel. Transitioning Legacy Assets to a Product Line Architecture. In *Proceedings of the Seventh European Software Engineering Conference (ESEC'99)*, LNCS 1687, pages 446-463, Toulouse, France, September 1999. Springer.
- [8] Bergey, J., L. O'Brien, and D. Smith. Mining Existing Assets for Software Product Lines. *Technical Note CMU/SEI-2000-TN-008*, Software Engineering Institute, Carnegie Mellon University, May 2000.
- [9] Biggerstaff, T.J. and A.J. Perlis. *Software Reusability: Concepts and Models*, Vol. I. Addison-Wesley, 1989.
- [10] Blok, M.C. and J.L. Cybulski. Reusing UML Specifications in a Constrained Application Domain. *Proc. 5th Asia Pacific Software Engineering Conference (ASPEC'98)*, pp. 196-202. Dec. 2-4, 1998.
- [11] Booch, G., J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Reading, MA: Addison-Wesley, 1998.
- [12] Bosch, J. Software Product Lines: Organizational Alternatives. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 91-100. IEEE Computer Society Press, Nov. 2001.
- [13] Bouchachia, A. and R.T. Mittermeir. Coping with Uncertainty in Software Retrieval Systems. *Proc. of the 2nd Intl. Workshop on Soft*

- Computing Applied to Software Engineering (SCASE'01)*, pp. 21-30, Feb. 2001.
- [14] Budanitsky, A. and Hirst G. Semantic Distance in WordNet: An Experimental, Application-Oriented Evaluation of Five Measures. *Proc. Workshop on WordNet and Other Lexical Resources*, Second Meeting of the North American Chapter of the Association of Computational Linguistics, Pittsburgh, PA, 2001.
 - [15] Budanitsky, A. Lexical Semantic Relatedness and Its Application in Natural Language Processing. *Technical Report CSRG-390*, Computer Systems Research Group, University of Toronto, August 1999.
 - [16] Burkard, R.E. and E. Cela. Linear Assignment Problems and Extensions. *Handbook of Combinatorial Optimization*. Kluwer Academic Publishers, pp. 75-149, 1999.
 - [17] Clements, P. and L. Northrop. A Framework for Software Product Line Practice, Version 3.0 [online]. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. <<http://www.sei.cmu.edu/plp/framework.html>>.
 - [18] Cybulski, J.L. and K. Reed. Requirements Classification and Reuse: Crossing Domain Boundaries. *Proc. 6th International Conference on Software Reuse*, Vienna, Austria, June 27-29, 2000.
 - [19] Cybulski, J.L. Introduction to Software Reuse. *Technical Report TR96/4*, University of Melbourne, Melbourne, Australia, July 1996.
 - [20] Cybulski, J.L. Personal Email Communication. May 25, 2002.
 - [21] Czarnecki, K., R. Hanselmann, U. W. Eisenecker, and W. Köpf. "ClassExpert: A Knowledge-Based Assistant to Support Reuse by Specialization and Modification in Smalltalk." In *Proceedings of the Fourth International Conference on Software Reuse*, Orlando, Florida, 1996, Murali Sitaraman (Ed.), IEEE Computer Society Press, 1996, pp. 188-194.
 - [22] Damiani, E., M.G. Fugini and C. Ballettini. A Hierarchy-Aware Approach to Faceted Classification of Object-Oriented Components. *ACM Trans. and Software Engineering and methodology*, 8(3):215-262, July 1999.
 - [23] De Champeaux, D. *Object-Oriented Development Process, and Metrics*. Prentice-Hall, Sept. 1996.
 - [24] De Mey, V. and O. Nierstrasz. The ITHACA Application Development Environment. *Visual Objects* (ed. D. Tsichritzis), Centre Universitaire d'Informatique, University of Geneva, July 1993, 297-280.
 - [25] Drummond, C.G., D. Ionescu, and R.C. Holte. A Learning Agent that Assists the Browsing of Software Libraries. *IEEE Transactions on Software Engineering*, Vol. 26, No. 12, Dec. 2000.

- [26] Edwards, S.H. The State of Reuse: Perceptions of the Reuse Community. *Software Engineering Notes*. Vol. 24, no. 3, May 1999, pp. 32 – 36.
- [27] Eisenbarth, T. and D. Simon. Guiding Feature Asset Mining for Software Product Line Development. In *Proc. of the International Workshop on Product Line Engineering - The Early Steps: Planning, Modeling, and Managing (PLEES'01)*. IESE-Report No. 050.01/E. Sept. 2001.
- [28] El-Emam, K. A Methodology for Validating Software Product Metrics. National Research Council of Canada, NRC/ERB 1076, June 2000.
- [29] Fellbaum, C. (Ed.). *WordNet: An Electronic Lexical Database*. MIT Press, 1998.
- [30] Fenton, N. Software Measurement: A Necessary Scientific Basis. *IEEE Transactions on Software Engineering*, 20(3), Mar. 1994.
- [31] Fenton, N.E. and S.L. Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. PWS Publishing Co. 1997.
- [32] Finkelstein, A., J. Kramer, and M. Goedicke. ViewPoint Oriented Software Development. *Proc. of 3rd Int. Workshop on Software Engineering and its Applications*, Toulouse, December 1990.
- [33] Fischer, B., "Specification-Based Browsing of Software Component Libraries", *Proc. 13th IEEE Conf. on Automated Software Engineering (ASE'98:)*, Honolulu, Hawaii, 1998, pp. 74-83.
- [34] Gamma, E., R. Helm , R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
- [35] Girardi, M.R. and B. Ibrahim. Using English to Retrieve Software. *Journal of Systems and Software*, 30(3): 249-270, September 1995. (Special Issue on Software Reuse)
- [36] Grand, M. *Patterns in Java, Volume I: A Catalog of Reusable Design Patterns*. John Wiley & Sons, 1998.
- [37] Grier, S. A Tool that Detects Plagiarism in Pascal Programs, *Twelfth SIGCSE Technical Symposium on Computer Science Education*, pp. 15-20, 1981.
- [38] Griss, M. Product-Line Architectures. In *Component-Based Software Engineering*, Edited by George T. Heineman and William Councill, Addison-Wesley, May 2001,
- [39] Griss, M. Software Reuse: From Library to Factory. *IBM Systems Journal* 32(4): 548-566 (1993).
- [40] Halstead, M.H. *Elements of Software Science*, Elsevier, North Holland, New York, 1977.

- [41] Hirst, G. and D. St-Onge. Lexical Chains as Representations of Context for the Detection and Correction of Malapropism. In [29], pp. 305 – 332.
- [42] Jiang, J. and Conrath, D. Multi-word complex concept retrieval via lexical semantic similarity *Proceedings International Conference on Information Intelligence and Systems*, 1999, pp. 407- 414
- [43] Jilani, L., J. Desharnais, and A. Mili. Defining and Applying Measures of Distance between Specifications. *IEEE Transactions on Software Engineering*. Vol. 27, No. 8, pp. 673-703; August 2001.
- [44] Kakeshita, T. and M. Murata. Specification-Based Component Retrieval by Means of Examples. *Proc. 1999 Intl. Symposium on Database Applications in Non-Traditional Environments (DANTE'99)* November 28-30, 1999 Kyoto, Japan, pp. 143-151.
- [45] Keller, R.K., J.F. Bédard, and G. Saint-Denis. Design and Implementation of a UML-Based Design Repository. *Proc. 13th International Conference on Advanced Information Systems Engineering (CAiSE2001)*, Interlaken, Switzerland, June 4-8, 2001.
- [46] Kitchenham, B., S.L. Pfleeger and N. Fenton. Reply to: Comments on “Towards a Framework for Software Measurement Validation.” *IEEE Transactions on Software Engineering*, 23(3), Mar. 1997.
- [47] Kitchenham, B., S.L. Pfleeger and N. Fenton. Towards a Framework for Software Measurement Validation. *IEEE Transactions on Software Engineering*, 21(12), Dec. 1995.
- [48] Klement, P. and W. Slany. Fuzzy Logic in Artificial Intelligence. *CD-Technical Report 94/67*, Technical University of Vienna, Austria, June 1994.
- [49] Kobryn, C. UML 2001: A Standardization Odyssey. *Communications of the ACM*, Vol. 42, No. 10, October, 1999, pp. 29-37.
- [50] Kruchten, P.B. The 4+1 View Model of Architecture. *IEEE Software*, Nov. 1995, pp. 42-50.
- [51] Krueger, C.W. Software Reuse. *ACM Computing Surveys*, 24(2), June 1992.
- [52] Kuhn, H.W. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly* 2, March-June 1955, pp 83-97.
- [53] Kukich, K. Technique for automatically correcting words in text, *ACM Computing Surveys (CSUR)*, v.24 n.4, p.377-439, Dec. 1992.
- [54] Leacock, C., and Chodorow, M., 1998. Combining Local Context and WordNet Similarity for Word Sense Identification. In C. Fellbaum (ed.) *WordNet: An Electronic Lexical Database*, 265-283. Cambridge, Mass: MIT press.

- [55] Lee, H-Y. and M.T. Harandi, An Analogy-Based Retrieval Mechanism for Software Design Reuse. *Proc. of the 8th Knowledge-Based Software Engineering Conference (KBSE'93)*, pp. 152-159, Chicago, 1993.
- [56] Lee, H-Y. *Automated Acquisition and Refinement of Reusable Software Design Components*. PhD Dissertation, University of Illinois at Urbana-Champaign, 1992.
- [57] Lin, D. 1998. An Information-Theoretic Definition of Similarity. In *Proceedings of the Fifteenth International Conference on Machine Learning*. Madison, Wisc: Morgan Kaufmann.
- [58] Lott, C.M. and H.D. Rombach. Repeatable Software Engineering Experiments for Comparing Defect-Detection Techniques. *Journal of Empirical Software Engineering*, Spring 1997.
- [59] Luqi and J. Guo, Toward automated retrieval for a software component repository. *Proc. IEEE Conf. and Workshop on the Engr. of Computer-Based Systems (ECBS '99)*, 1999 pp.: 99 -105.
- [60] McGregor, J. and D. Sykes. *Object-Oriented Development: Engineering for Software for Reuse*. Van Nostrand, 1992.
- [61] McIlroy, M. D., Mass produced software components, *Proc. NATO Software Eng. Conf.*, Garmisch, Germany (1968) 138-155. Also available at <http://www.cs.dartmouth.edu/~doug/components.txt>
- [62] Meling, R., E.J. Montgomery, P.S. Ponnusamy, E.B. Wong, and D. Mehandjiska. Storing and Retrieving Software Components: A Component Description Manager. *Proc. of the 2000 Australian Software Engineering Conf.*, Canberra, Australia, April 2000, pp. 107 --117.
- [63] Mendenhall, W. and T. Sincich. *Statistics for Engineers and the Sciences*. Prentice-Hall, 1995.
- [64] Michail, A. and D. Notkin. Assessing Software Libraries by Browsing Similar Classes, Functions and Relationships. *Proc. 21st International Conference on Software Engineering*, Los Angeles, 1999, pp 463-472.
- [65] Mili, A., R. Mili, and R. Mittermeir. A survey of Software Reuse Libraries. *Annals of Software Engineering*, 1998.
- [66] Mili, H., F. Mili and A. Mili. An Introduction to Software Reuse. *Technical Report ISR-98-08-23-MMM*, August 23, 1998.
- [67] Mili, H., F. Mili and A. Mili. Reusing Software: Issues and Research Directions, *IEEE Transactions on Software Engineering*, Vol. 21, No. 6, June 1995.
- [68] Mili, H., O. Marcotte, and A. Kabbaj. Intelligent Component Retrieval for Software Reuse. *Proceedings of the Third Maghrebian Conference on Artificial Intelligence and Software Engineering*, pp. 101-114, Rabat, Morocco, April 11-14, 1994.

- [69] Mili, R., A. Mili, and R.T. Mittermeir. Storing and Retrieving Software Components: A Refinement Based System. *IEEE Trans. On Software Engineering*, Vol. 23, No.7, July 1997.
- [70] Miller, J. Replicating software engineering experiments: a poisoned chalice or the Holy Grail. [Draft Book Chapter]. Oct. 2000. Available at <http://sern.ucalgary.ca/courses/SENG/693/F00/readings/JamesMiller.pdf> [Accessed 2002-12-10].
- [71] Morasca, S., L.C. Briand, V.R. Basili, E.J Weyuker and M.V. Zelkowitz. Comments on "Towards a Framework for Software Measurement Validation." *IEEE Transactions on Software Engineering*, 23(3), Mar. 1997.
- [72] Northrop, L.M. SEI's Software Product Line Tenets. *IEEE Software*, 19(4):32-40, July/August 2002.
- [73] Nuseibeh, B., J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specification, *IEEE Transactions on Software Engineering*, Vol.20, Iss.10, 1994, pp. 760- 773.
- [74] Objects by Design, Inc. Transforming XMI to HTML. http://www.objectsbydesign.com/projects/xmi_to_html.html (Accessed 2002-12-20).
- [75] Oestereich, B. *Developing Software with UML- Object-Oriented Analysis and Design in Practice (2nd Ed.)*. Addison-Wesley, 2001.
- [76] OMG. *Introduction to OMG's Unified Modeling Language*. http://www.omg.org/gettingstarted/what_is_uml.htm (Accessed 2002-07-01).
- [77] OMG. *OMG Unified Modeling Language Specification (Version 1.4)*. Object Management Group (OMG). Sept. 2001. Available at <http://www.omg.org/technology/documents/formal/uml.htm> (Accessed 2002-07-01).
- [78] OMG. *OMG XML Metadata Interchange (XMI) Specification (Version 1.2)*. Object Management Group (OMG). Jan. 2002. <http://www.omg.org/technology/documents/formal/xmi.htm> (Accessed 2002-06-01).
- [79] Ottenstein, K.J. An Algorithmic Approach to the Detection and Prevention of Plagiarism, *SIGCSE Bulletin*, Vol. 8 No. 2, Dec. 1976.
- [80] Pedersen, T. and S. Patwardhan. Semantic Distance Measures Perl utility version 0.11 (distance-0.11) Readme, University of Minnesota, Duluth, 2002. <http://www.d.umn.edu/~tpederse/Code/Readme.distance-0.11.txt> (Accessed 2002-12-10)

- [81] Penix, J. and P. Alexander: Using Formal Specifications for Component Retrieval and Reuse. *Proc. 31st Annual Hawaii International Conference on System Sciences (HICSS)*, 1998, pp. 356-365.
- [82] Pozewaunig, H. and R.T. Mittermeir. Self Classifying Reusable Components: Generating Decision Trees from Test Cases. *Proc. International Conference on Software Eng. and Knowledge Eng.* Chicago, IL., July 6-8, 2000.
- [83] Prechelt, L., G. Mahpohl, and M. Phlippsen. Jplag: Finding Plagiarisms among a set of Programs. *Technical Report 2000-1*, Universitat Karlsruhe, March 2000.
- [84] Prieto-Diaz, R. Status Report – Software Reusability. *IEEE Software*, 10(3):61-66, May 1993.
- [85] Resnik, P. Using information content to evaluate semantic similarity in a taxonomy. *Proceedings of IJCAI-95*, Montreal, Canada, 1995, pp. 448–453.
- [86] Rich, C. and R. C. Waters. "The Programmer's Apprentice: A Research Overview", *IEEE Computer*, vol. 21, no. 11, Nov. 1988, pp. 11-25.
- [87] Rine, D.C. Success factors for software reuse that are applicable across domains and businesses. *Proc. of the ACM Symposium on Applied Computing*, 1997: 182-186.
- [88] Rufai, R., M. Ahmed, and J. AlGhamdi. Towards a Unified Software Component Representation. *Proc. SCI-2002*, Orlando, FL, July 2002, pp. 146-150.
- [89] Salton, G. and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill. 1983: 118 – 156.
- [90] Santini, S. and R. Jain. Similarity Measures. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Sept. 1999.
- [91] Sarireta, A. and J. Vaucher. Similarity Measure in the Object Model. *Proc. ECOOP'97*, Jyvaskyala, Finland, June 9-13, 1997.
- [92] Schmid, K. A Comprehensive Product Line Scoping Approach and Its Validation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 593–603, May 2002.
- [93] Schneiderwind, N.F. Methodology for Validating Software Metrics. *IEEE Transactions on Software Engineering*, 18(5), Mar. 1992.
- [94] Schumann, J. and B. Fischer: NORA/HAMMR: Making Deduction-Based Software Component Retrieval Practical. *Proc. Automated Software Engineering (ASE-97)*, Lake Tahoe, November 1997, pp. 246-254.
- [95] Simon, F., S. Löffler, and C. Lewerentz. Distance-Based Cohesion Measuring. *FESMA99*, Amsterdam, 4-8 October, 1999.
- [96] Sommerville, I. *Software Engineering*, 6th Ed. Addison-Wesley, 2000.

- [97] St-Onge, D. Detecting and Correcting Malapropism with Lexical Chains. MS Thesis. Department of Computer Science, University of Toronto, Toronto, Canada, March 1995.
- [98] Tversky, A. Features of Similarity. *Psychological Review*. 84:327-352, 1977.
- [99] Verco, K.K. and M.J. Wise. Software for Detecting Suspected Plagiarism: Comparing Structure and Attribute-Counting Systems. *Proc. of 1st Australian Conference on Computer Science Education*, Sydney, July 1996.
- [100] Wegner, P. Capital-Intensive Software Technology. *IEEE Software*, 1(3), July 1984.
- [101] Whale, G. Identification of Program Similarity in Large Populations. *The Computer Journal*, 33(2):140-146, 1990.
- [102] Wilson, W.M., L.H. Rosenberg, and L.E. Hyatt, Automated Quality Analysis of Natural Language Requirement Specifications, *Proc. Pacific Northwest Software Quality Conference*, Oct. 1996, pp. 140-151 http://satc.gsfc.nasa.gov/support/PNSQC_OCT96/png.PDF (Accessed 2002-12-10).
- [103] Yacoub, S.M.. Composite Filter Pattern, *Proc. EuroPLoP 2001*, Irsee, Germany, 4-8 July 2001.
- [104] Ye, Y. *Supporting Component-Based Software Development with Active Component Repository Systems*. PhD Dissertation, University of Colorado, Boulder, CO, 2001.
- [105] Zaremski, A. M. and J. M. Wing. Signature Matching: a Tool for Using Software Libraries. *ACM Transactions on Software Engineering and methodology*, 4(2):146-170, April 1995.
- [106] Zaremski, A.M. and J.M. Wing. Specification Matching of Software Components. *ACM Transactions and Software Engineering and methodology*, 6(4):333-369, October 1997.

Index

- activity diagram, 39
- ANT, viii, 77, 79, 82, 83, 84, 85, 86, 87,
91, 92, 93, 94, 96, 97, 101, 102,
104, 106, 107, 109, 111
- application engineering, 9
- asset mining, 10, 11
- behavioral view, 36, 38, 39
- building blocks approach, 3
- cascading, 40, 41, 54, 57, 63, 105, 114
- class diagrams, 36, 39, 43
- class model, 18, 23, 36, 43, 47, 48, 74,
112
- classification trees, 28, 115
- collaboration diagrams, 36
- component diagram, 39
- composite filter pattern, 63
- composite pattern, 63
- composition, 35, 40
- contrast model, 38
- cosine similarity measure, 20
- decision trees, 27
- deduction-based retrieval, 24
- deployment diagram, 36, 39
- deployment diagrams, 36
- design repository, 4
- development process, 3, 6, 29
- DFD diagram, 16, 23
- domain engineering, 9
- DSSM, vi, 43, 51, 52, 53, 55, 60, 67, 68,
73, 74, 75, 76, 110, 114
- early-stage reusable artifacts, 5
- empirical validation, 65, 71, 73
- ER diagram, 16
- factory metaphor, 9
- filter pattern, 63
- functional similarity, 25
- functional specification, 24, 26
- fuzzy sets, 18
- generalization, 27, 28, 115
- Hungarian algorithm, 50
- hypothesis, 73, 75, 78, 79, 110
- implementation view, 36, 39
- inconsistency penalty, 38, 39, 40, 41, 42,
113
- information theory, 45
- intuition, 48, 65, 71, 72, 73, 74, 76, 78,
83, 88, 92, 94, 98, 102, 108,
110, 112
- Java, 105, 114, 118
- Java Software Development Kit, viii, 79,
80, 81, 82, 83, 84, 85, 89, 90,
91, 92, 93, 94, 99, 100, 101,
102, 103, 104, 109, 111
- JPlag, 15
- later-stage reusable artifacts, 5
- library metaphor, 9
- matching, 13, 14, 18, 19, 20, 21, 23, 24,
25, 26, 28, 36, 41, 43, 51, 52,
60, 73, 112
- meta-modeling, 31, 32, 33
- metric axioms, 65
- multi-view similarity, iii, 8, 12

- naming malpractices, 54
- NATO software engineering conference, 1
- object diagram, 39
- Object Management Group, 30, 31, 32, 33, 121
- Object Modeling Technique, 29
- Objectory (OOSE), 29
- Perl, 46, 113, 121
- Plan Calculus, 24
- process, 1, 2, 3, 6, 8, 10, 16, 18, 26, 29, 77
- product family, 10
- proportionate change, viii, 83, 84, 85, 92, 93, 94, 102, 103, 104
- RBSM, vi, viii, 58, 59, 60, 70, 73, 74, 75, 76, 79, 99, 100, 101, 102, 103, 104, 106, 107, 108, 109, 110
- recall, 26
- representation condition, 71, 72
- requirements specification, 3, 4, 27, 121
- retrieval engine, 54, 57
- retrieval systems, 3, 7, 11, 13
- reusable artifacts, 2, 3, 5, 6, 7, 10, 17
- reusable processor approach, 3
- reusable software, 1, 3, 17, 30
- SBSM, vi, viii, 54, 55, 57, 60, 69, 70, 73, 74, 75, 76, 79, 89, 90, 91, 92, 93, 94, 96, 97, 98, 109, 110, 115
- scoping, 10, 122
- semantic distance, 21, 44, 45, 46, 62, 113, 114
- semantic network, 50, 53
- semantic relatedness, 44
- semantic relation, 18
- sequence diagram, 20, 23, 39, 41, 113
- signature matching, 24
- similarity metric, iii, 7, 8, 11, 12, 21, 35, 38, 39, 40, 42, 43, 47, 65, 73, 76, 79, 83, 92, 102, 105, 110, 113
- similarity metrics, iii, 7, 8, 11, 12, 21, 35, 38, 39, 40, 42, 43, 47, 65, 73, 76, 79, 83, 92, 102, 105, 110, 113
- software product lines, 4, 9, 10, 112
- software repository, 7
- software retrieval, 3, 7, 112
- software reuse, 1, 2, 3, 4, 6, 8, 9, 73, 122
- software reuse process, 2, 3
- Spearman rank correlation, 78
- specialization, 27, 115
- SSSM, vi, viii, 43, 47, 48, 49, 50, 52, 59, 60, 62, 66, 67, 73, 75, 76, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 109, 110, 114
- state-chart diagrams, 36, 39
- strategy pattern, 63
- string matching, 23, 56
- structural similarity, iii, 7, 12, 25, 38
- structural view, 36, 39, 62
- surrogates, 7
- test data, 4, 27
- theoretical validation, 65
- theory of measurement, 71
- tools, vi, 77, 118, 123, 127

- GD-Pro, 33
- Microsoft Visual Modeler, 33
- Object Domain, 33
- Object Team, 33
- Rational Rose, 33, 34, 77
- System Architect, 33
- Together J, 33, 77
- Visio, 33
- Visual UML, 33
- traceability, 11, 19
- UML models, 5, 7, 11, 12, 20, 43, 72, 73, 74, 76, 77, 112, 115
- Unified Modeling Language, iii, iv, v, 7, 11, 12, 13, 20, 23, 29, 30, 31, 32, 33, 34, 36, 39, 43, 47, 52, 72, 73, 74, 75, 76, 77, 112, 113, 115, 116, 119, 121
- use-case view, iii, 36, 39, 113
- VDM-SL, 25
- viewpoint-oriented approach, 35, 36
- WordNet, viii, 18, 44, 45, 117, 118, 119
- XMI, 32, 33, 34, 77, 115, 121

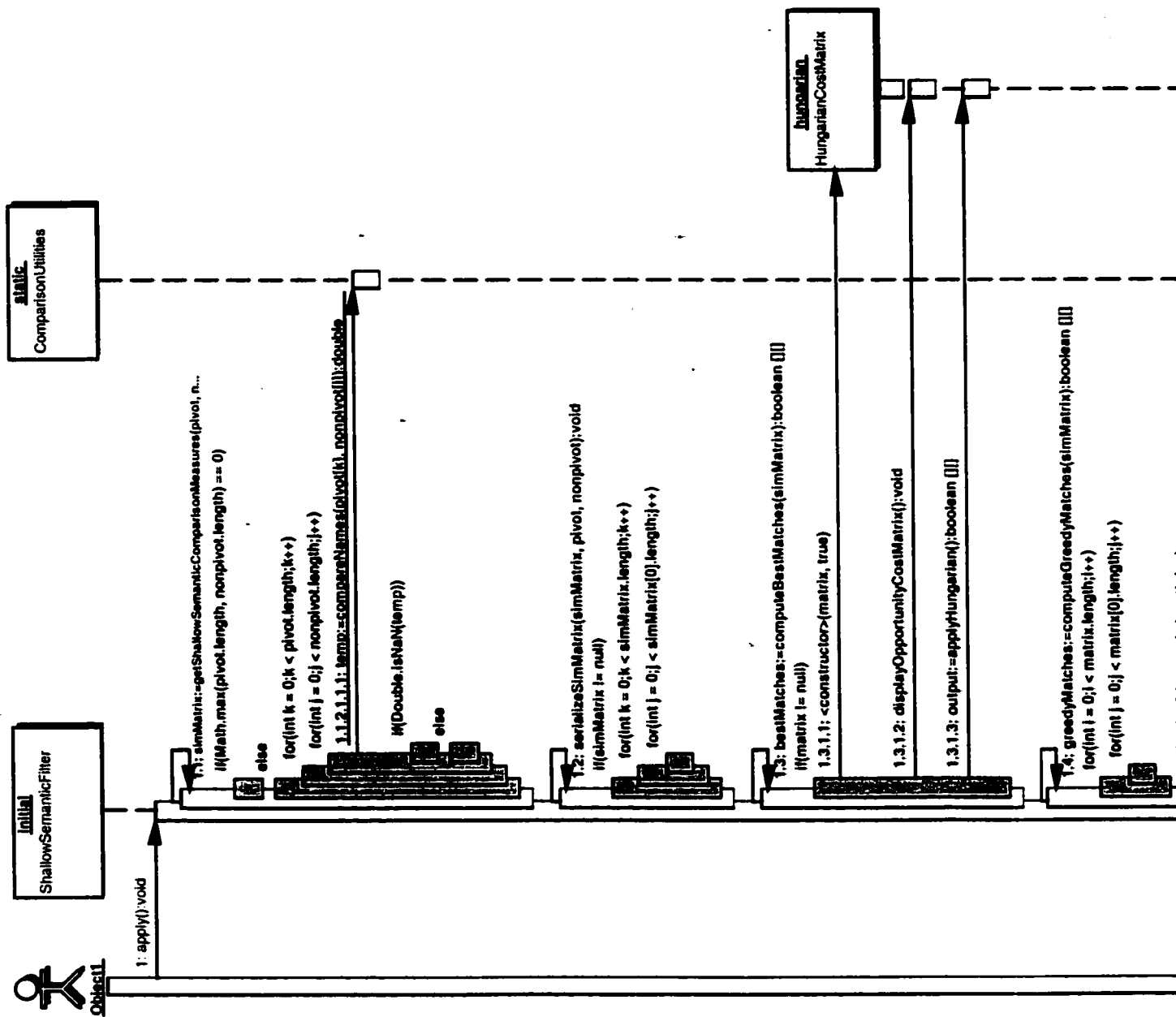
Appendix A: Tool Design

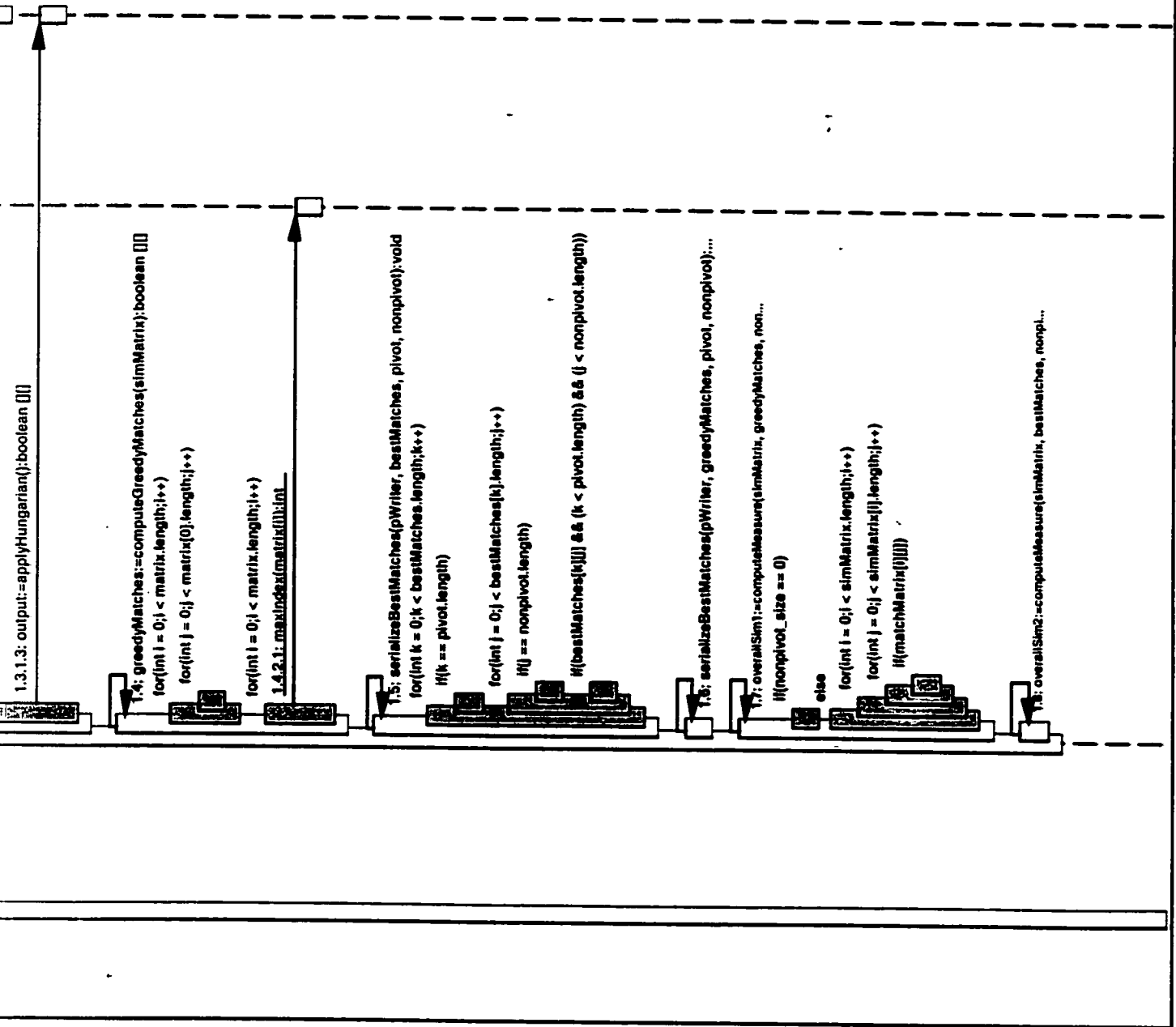
The diagram shows a complex set of classes and interfaces. Key components include:

- JPanel**: A large class with many attributes (e.g., `ModelPair`, `Model1`, `Model2`) and methods (e.g., `generatePairings`, `generateCDPairing`, `generateSDPairing`, `generateUCDPairing`).
- JFrame**: Contains a `MainWindow` attribute and methods like `borderLayout`, `panel1`, `menuBar`, `menuLibraryAdd`, `menuLibraryDelete`, `menuLibrarySearch`, `menuLibraryBrowse`, `menuLibraryPreferences`, `menuLibraryExit`, `menuModelCompare`, `menuModelHistory`, `mainWindow`.
- JFileChooser**: Contains `FileFilter` and `UIFilter` attributes and methods like `accept`, `description`.
- Interfaces**: `Metric` (with `ACD_EXTENSION`, `ASD_EXTENSION`, `UCD_EXTENSION`), `ConsistencyFactor` (with `computeMeasure`), and `ConsistencyType` (with `computeMeasure`).
- Concrete Classes**: `CDSDM`, `UCDSDM`, `SDSDM`, `ModelPair`, `Model1`, `Model2`, `Model3`, `Model4`, `Model5`, `Model6`, `Model7`, `Model8`, `Model9`, `Model10`, `Model11`, `Model12`, `Model13`, `Model14`, `Model15`, `Model16`, `Model17`, `Model18`, `Model19`, `Model20`, `Model21`, `Model22`, `Model23`, `Model24`, `Model25`, `Model26`, `Model27`, `Model28`, `Model29`, `Model30`, `Model31`, `Model32`, `Model33`, `Model34`, `Model35`, `Model36`, `Model37`, `Model38`, `Model39`, `Model40`, `Model41`, `Model42`, `Model43`, `Model44`, `Model45`, `Model46`, `Model47`, `Model48`, `Model49`, `Model50`, `Model51`, `Model52`, `Model53`, `Model54`, `Model55`, `Model56`, `Model57`, `Model58`, `Model59`, `Model60`, `Model61`, `Model62`, `Model63`, `Model64`, `Model65`, `Model66`, `Model67`, `Model68`, `Model69`, `Model70`, `Model71`, `Model72`, `Model73`, `Model74`, `Model75`, `Model76`, `Model77`, `Model78`, `Model79`, `Model80`, `Model81`, `Model82`, `Model83`, `Model84`, `Model85`, `Model86`, `Model87`, `Model88`, `Model89`, `Model90`, `Model91`, `Model92`, `Model93`, `Model94`, `Model95`, `Model96`, `Model97`, `Model98`, `Model99`, `Model100`.



1.1.1, ShallowSemanticFilter.apply(4)





1.1, ShallowSemanticFilter.apply(4)

1.1, DeepSemanticFilter.apply(I)

static
ComparisonUtilities

Initial
DeepSemanticFilter

Object1

1: apply():void

1.1: serializeSimMatrix(simMatrix, pivot, nonpivot):void
if(simMatrix != null)
for(int k = 0; k < simMatrix.length; k++)
for(int j = 0; j < simMatrix[0].length; j++)

1.2: bestMatches:=computeBestMatches(simMatrix):boolean [][]
if(matrix != null)
1.2.1.1: <constructor>(matrix, true)
hungarian_ HungarianCostMatrix

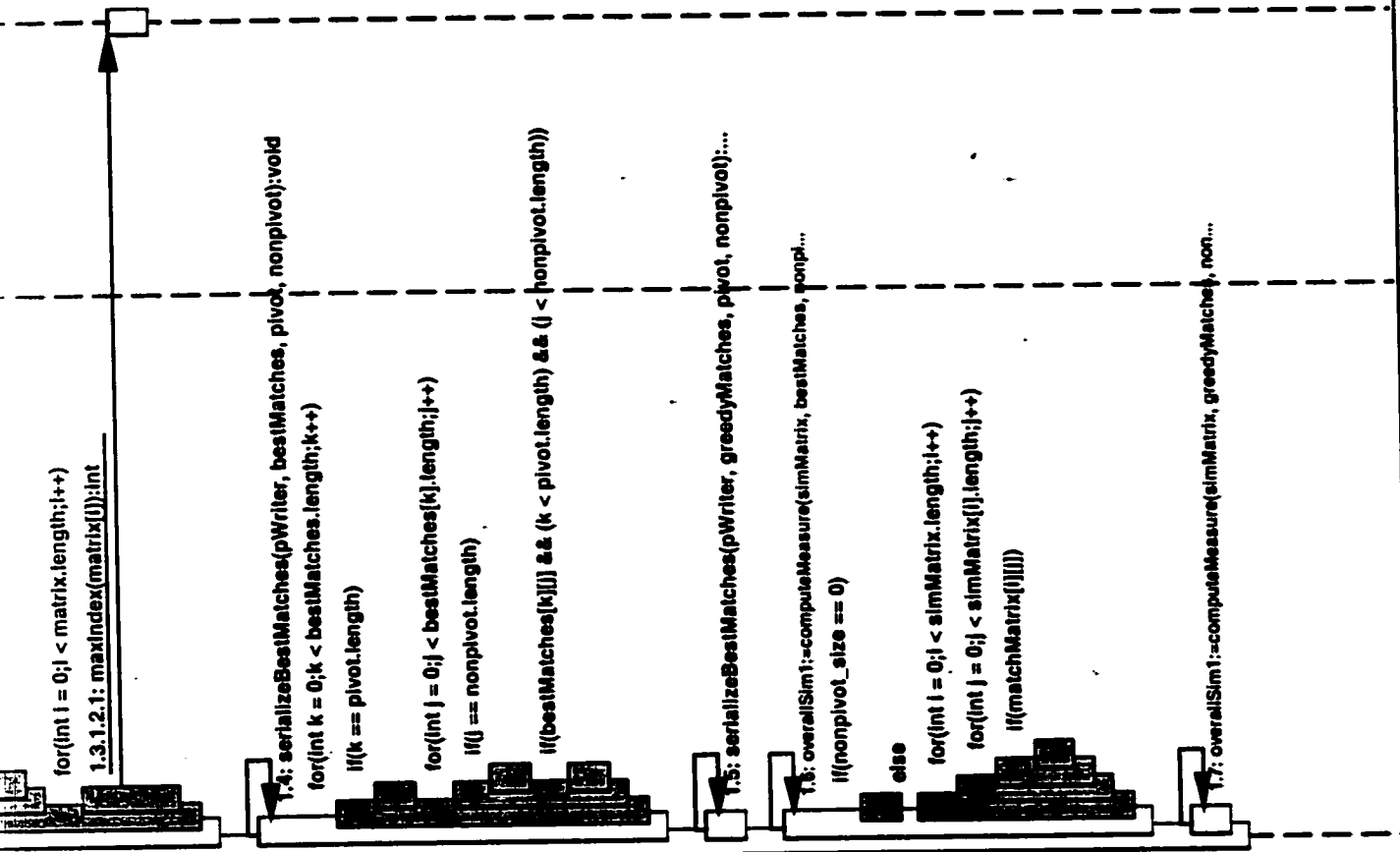
1.2.1.2: displayOpportunityCostMatrix():void

1.2.1.3: output:=applyHungarian():boolean [][]

1.3: greedyMatches:=computeGreedyMatches(simMatrix):boolean [][]
if(matrix != null & matrix.length > 0)
for(int i = 0; i < matrix.length; i++)
for(int j = 0; j < matrix[0].length; j++)

for(int i = 0; i < matrix.length; i++)
1.3.1.2.1: maxIndex(matrix[i]):int

1.4: serializeBestMatches(pWriter, bestMatches, pivot, nonpivot):void



1.1, DeepSemanticFilter.apply(1)

1.1, SignatureFilter.apply(1)

static
ComparisonUtilities

Initial
SignatureFilter

Object

1: apply():void

1: serializeSimMatrix(simMatrix, pivot, nonpivot):void
if(simMatrix != null)
for(int k = 0; k < simMatrix.length; k++)
for(int j = 0; j < simMatrix[0].length; j++)

1.2: bestMatches:=computeBestMatches(simMatrix):boolean [][]
if(matrix != null)
1.2.1.1: <constructor>(matrix, true)

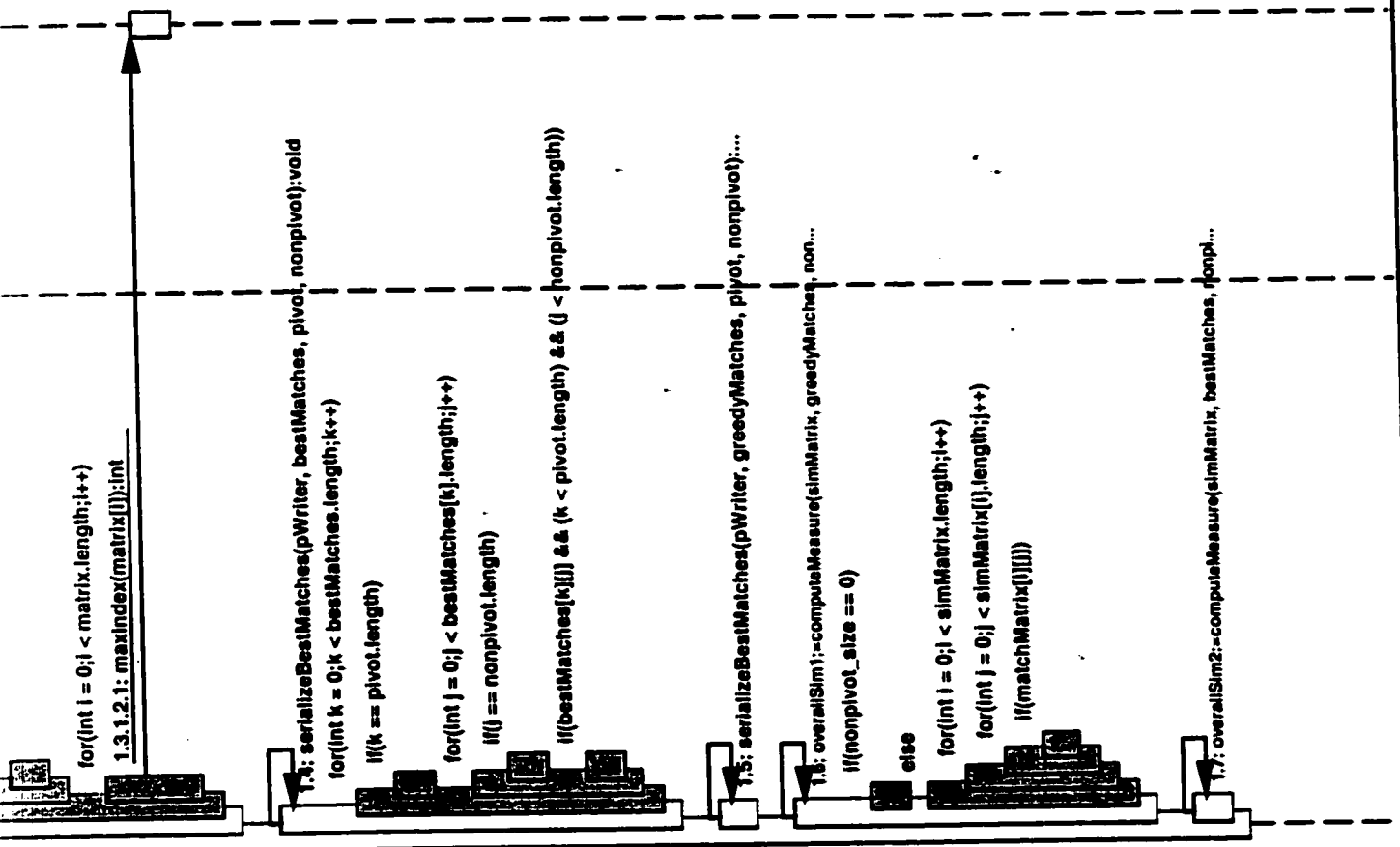
hungarian
HungarianCostMatrix

1.2.1.2: displayOpportunityCostMatrix():void

1.2.1.3: output:=applyHungarian():boolean [][]

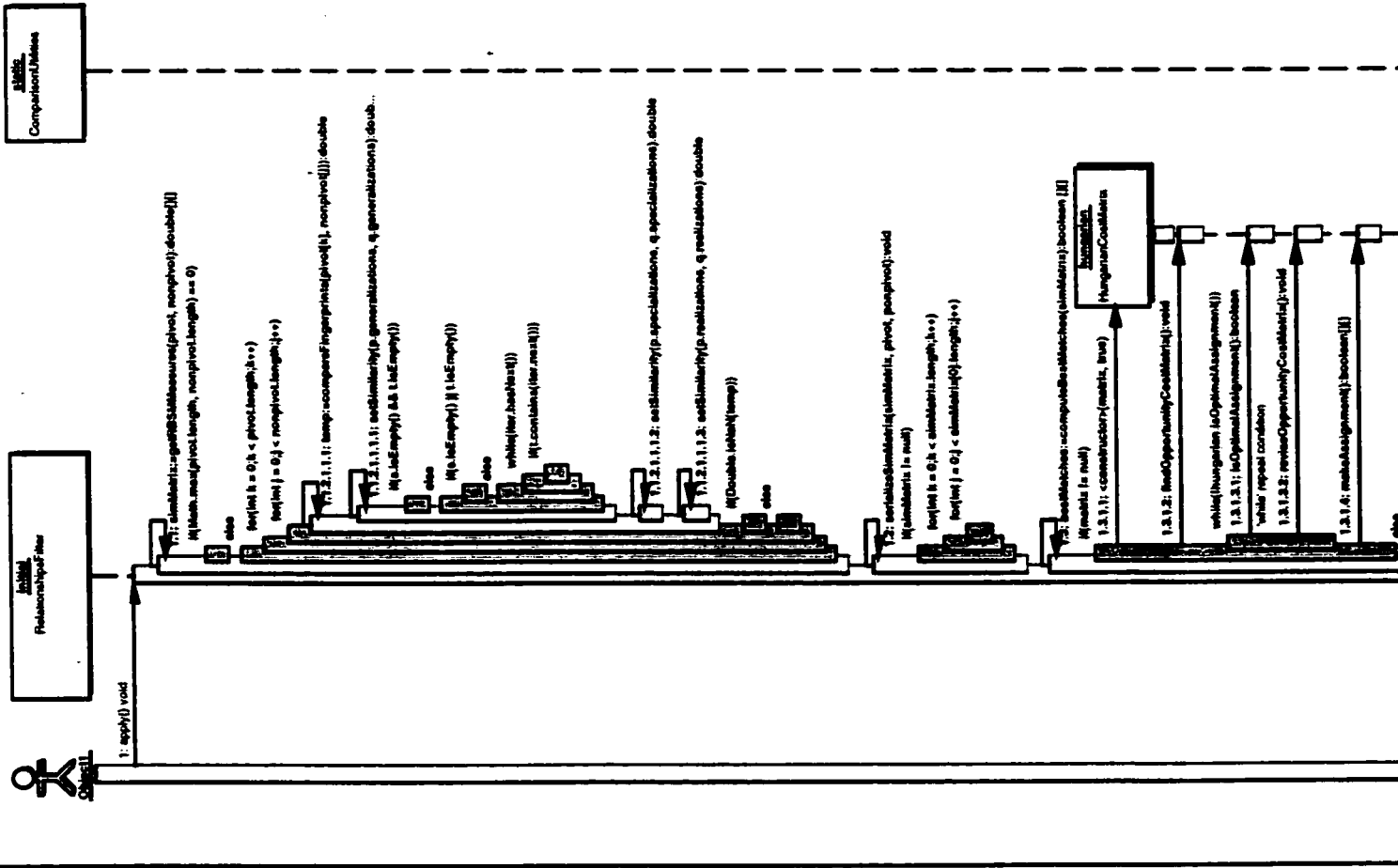
1.3: greedyMatches:=computeGreedyMatches(simMatrix):boolean [][]
if(matrix != null && matrix.length > 0)
for(int i = 0; i < matrix.length; i++)
for(int j = 0; j < matrix[0].length; j++)
for(int i = 0; i < matrix.length; i++)
1.3.1.2.1: maxIndex(matrix[i]):int

1.4: serializeBestMatches(pWriter, bestMatches, pivot, nonpivot):void



1.1, SignatureFilter.apply(1)

Basic Comparison Utilities



Vita

Raimi Ayinde Rufai, who hails from Shaki, Nigeria, obtained the B.Sc. degree in Computer Science from University of Ilorin, Ilorin, Nigeria in 1997. Prior to attending King Fahd University of Petroleum and Minerals (KFUPM), he worked in Sheladia/Yolas & Associates, Abuja, Nigeria, from March 1998 to August 1998 on the World Bank funded “Nigerian National Road Network Study” project as a Programmer. He then moved to iTECO Nigeria Ltd. (now called SoftWorks Ltd.), where he worked as a Programmer/Analyst until August, 2000. In August 2000, Mr. Rufai joined KFUPM as a Research Assistant to pursue the MS Computer Science degree. Mr. Rufai’s research interests include software reuse, software metrics, OO software modeling, UML, XML, and software process modeling and process improvement. He can be reached at rrufai@acm.org.