Toward Experimental Evaluation of Subsystem Classification Recovery Techniques

Arun Lakhotia and John M. Gravley
Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504
{arun, jmg}@cacs.usl.edu
(318) 482-6766, Fax: -5791

Published in: Proc. of the 2nd Working Conference on Reverse Engineering, © IEEE CS Press, July 1995

Abstract

Several reverse engineering techniques classify software system components into subsystems. These techniques are designed to discover such classifications when the classifications are unknown. The techniques are tested and evaluated, however, by matching the classifications they recover against expected classifications. Several such techniques may be compared by experimentally evaluating their performance on the same set of software systems. Two things are needed to ensure experiment repeatability: (1) a set of "real-world" software systems whose expected subsystem classifications are known, and (2) an objective criterion to quantitatively determine the similarity of subsystem classifications. This paper contributes to both needs by identifying a set of widely used and easily accessible software systems whose modular decomposition either is documented or can be easily inferred from their design philosophy, and by presenting a measure to quantitatively determine the congruence between hierarchical subsystem classifications.

1 Introduction

Several reverse engineering techniques have been designed to classify the components of a software system into subsystems with the intent of aiding reengineering or program understanding. Such a technique, called a subsystem classification recovery technique (SCRT), takes as its input some interconnection relations between the components of a program and outputs a classification of these components into subsystems. The classification may be performed to discover the modular architecture of undocumented legacy code [15, 17, 18], or to group program components that are related to the same error [11], or simply to impose a structure upon an otherwise large, complex, and unstructured data [4, 16].

To date, at least fourteen techniques have been proposed for classifying software subsystems [14]. These techniques have been empirically evaluated by their designers, typically within the context of their laboratories or projects. This paper is part of our continuing effort to comparatively evaluate SCRTs. In an earlier work, the first author of this paper has presented a unified framework to describe SCRTs using a consistent terminology and notation [14]. This paper establishes some of the parameters necessary to validate and compare SCRTs using controlled experiments.

The subsystem classification (SC) problem belongs to the general problem of classifying the elements of a population. This class of problems has been studied for over 100 years, spanning the fields of the life sciences, the social sciences, medicine, the earth sciences, and engineering. In the context of computing, classification problems have been investigated in the realm of pattern recognition and information retrieval. There is, therefore, a rich body of literature investigating various aspects of classification problems. In spite of this fact, there is no classification technique that, if directly applied to a new classification problem, will guarantee good results. This is because classification techniques are essentially heuristic and rely upon knowledge specific to the problem domain.

There is no single, general purpose classification technique, but research into classification problems has led to classes of techniques, i.e., generic templates, that may be customized for individual applications. Researchers have also identified a set of generic properties that may be used to evaluate classification techniques. One such property that is also relevant for evaluating SCRTs is: How effective is a SCRT in recovering the *expected* classifications?

Evaluating the effectiveness of a SCRT is no different than evaluating the effectiveness of any other heuris-

tic technique. To do so, a program, or a technique, is subjected to several different inputs and its outputs are compared against the expected outputs. Testing the effectiveness of such techniques requires either the existence of a set of data for which the expected results are known or the existence of an *oracle*—a mechanism different from the program itself—that can provide the expected output [12]. A *comparator*—a mechanism to compare the output of a program and the expected output—is also required to evaluate the results of a test. When testing SCRTs, so far the roles of the oracle and the comparator have most often been combined into one by having the similarity between an output classification and the expected classification assessed by a *tester*, typically a system analyst, a maintainer of the subject program, or the designer of the SCRT. Such an assessment is typically done by having a tester compare an output classification with a "mental model" of the corresponding expected classification [16, 25].

A system analyst, a maintainer of a subject program, or the SCRT designer are valuable oracles, but not explicitly representing the classifications they expect renders such methods of testing non-repeatable. That is, it does not guarantee that the same tester would give the same assessment at a different time, or that two different testers would give the same result.

Even if the expected classifications were explicitly represented, using human testers as comparators also renders the results non-repeatable. This is because SCRTs, being heuristics, are not likely to produce precisely the expected classifications. Hence, a SCRT may be evaluated by how *similar* the classifications it recovers are to those expected. A manual assessment of the degree of similarity between classifications is subject to human error as the size of the classifications increase.

Repeatability is a fundamental requirement for scientific experiments. At the very minimum, experiments should be repeatable within a single laboratory. Ideally, they should be repeatable by different research groups. Until now, the evaluation of SCRTs has been rendered non-repeatable within a laboratory because: (1) the expected classifications were not explicitly represented, and (2) the comparison of the classification was not objective. SCRT evaluation has also been non-repeatable across multiple laboratories because they have most often been tested with subject programs that were proprietary, thus making the data inaccessible to other researchers.

To aid in the design of repeatable experiments for evaluating SCRTs that recover modular subsystems, this paper proposes: (1) a set of subject programs, and (2) a measure of congruence that may be used as a *comparator* of classifications. The proposed subject programs have the property that they are widely used, are easily accessible, and, above all, their expected subsystem classifications are known or can be easily inferred based upon their design philosophy. These programs have the potential to become benchmarks for evaluating SCRTs.

The rest of this paper is organized into the following sections. In Section 2 we discuss the proposed set of benchmark programs for evaluating SCRTs. In Section 3 we establish a measurement theory based framework for developing our congruence measure. In Section 4 we present our measure of congruence between SCs. Finally, in Section 5 we present our conclusions.

2 Proposed subject programs

Choosing subject programs for experimentally evaluating program analysis tools requires balancing at least three constraints. First, the subject programs should be representative of *real-world* programs in order to draw meaningful inferences from the data gathered. Second, there should exist *oracles* that give the expected results of the analyses against which the output of the tool may be evaluated. Third, the subject programs should be accessible to other researchers.

These three constraints are difficult to satisfy simultaneously, especially when the programs are to be used for the comparative evaluation of SCRTs. There are at least four reasons. First, there may not be a consensus among researchers on what constitutes a representative sample of real-world programs. For example, can a random sample of public domain programs be treated as a representative sample of proprietary programs? Second, there may not be a consensus on the *oracles* to be used. Third, assuming a consensus is achieved on the first two reasons, making progress toward satisfying one constraint may negatively affect the other. That is, we may find a random sample of real-world programs, but not have oracles that provide the results expected from analyzing them. Last, the analyses or tools implementing various SCRTs may have been designed for different programming languages, making it impossible to evaluate them on a common set of programs.

Such disagreements, trade-offs, or experimentation difficulties are common to experiments in any domain. They do not diminish the need for using standard test data, but they instead emphasize that careful attention should be paid to choosing such data, and that the results from any experiment using this data should be analyzed within the context of these limitations.

We have identified a set of software systems that may be used to evaluate SCRTs that recover from *C programs* subsystem classifications representing *modules*. Notice the emphasis on modules. The programs are *not* being proposed to evaluate SCRTs that recover other subsystem classifications. The program set is constrained to the evaluation of modular SCRTs because we have identified an oracle for each program that gives its expected mod-

	Class Assignments	*Class Projects*	*layered source real-world systems*	*scattered source real-world systems*
Size in LOC	500-2000	3000-5000	20,000+	20,000+
Specifications	Controlled	Controlled	Uncontrolled	Uncontrolled
Module hierarchy	Controlled	Not controlled Documented	Not controlled May be extracted	Not controlled Not obvious
Number of programs	3 sets of 15 to 20 programs (*)	3 sets of 5 to 10 programs (**)	5 to 10	3 to 5
Sample examples/problems	lexical analyzer expression evaluator graph traversal	hypertext software metrics room scheduler	SGEN, WPIS, FIELD, LSL/LCL	FSF programs Unix utilities

* All programs in a set implement the same specification and design.
** All programs in a set implement the same specification.
WPIS: Wisconsin Program Integration system from University of Wisconsin [10]
SGEN: Synthesizer Generator from GrammaTech, Inc. [22, 23]
FIELD: Friendly Integrated Environment for Learning and Development from Brown University [21]
LSL/LCL: Larch Shared Language and Larch C Interface Language from DEC Software Research Center [9, 8]
FSF: Free Software Foundation
Unix is a registered trademark of AT&T

Table 1. Summary of characteristics of proposed subject programs.

ule classification. We constrain ourselves to identifying C programs because C is currently a de facto standard in the industry. We believe this fact makes C a good choice for experimentation since today's new systems are tomorrow's legacy systems. The source code and documentation for all of the programs we identify is accessible either for free or for a fee. Hence, the only constraint that remains to be satisfied is whether the subject programs form a representative sample of real-world programs.

The fact that oracles exist that provide the modular classification of these systems implies that they are likely to be well-designed and hence *not* a representative sample of real-world programs. However, these subject programs may be representative of well-designed modular programs. We contend that a SCRT will be more effective in recovering modular subsystems for programs with better modular designs as opposed to ill-designed programs. Otherwise, it will violate the garbage-in, garbage-out principle. Assuming this contention, our subject programs may be used to establish upper bounds on the effectiveness of SCRTs.

We classify the proposed subject programs into four categories: class *assignments*, class *projects*, *layered-source* systems, and *scattered-source* systems. Table 1 summarizes some of the salient characteristics of these programs.

The class assignments and projects we propose as subject programs have been developed as part of a senior level software engineering course at the University of Southwestern Louisiana, Lafayette, LA. They have the characteristic that they can be partitioned into sets of size greater than one, with all programs in a set implementing the same specification. The use of class assignments and projects with identical specifications is common in software engineering experiments [2, 11]. We introduce two differences:

1. The class assignments implementing the same specification also implement the same design (a design provided by the instructor).
2. That the class assignments and projects adhere to the principles of modular design is ensured by performing a strict quality check consisting of design and code reviews.

Though the class assignments and projects are by no means representative of real-world programs, they are nonetheless valuable. They enable the studying of the effect on subsystem classifications due to variations in design and/or implementation decisions, while factoring out the effect of changes in specifications.

The last two categories of programs—layered-source and scattered-source—are indeed real-world programs. These programs definitely satisfy some important real-world need since they are being actively used and maintained. The categorization into layered-source and scattered-source is based upon the organization of the source code in their distribution kits. In the layered-source system, the code for each module is either placed

into a separate file [9, 8, 21] or in multiple files contained in a separate directory [10, 22, 23]. The directories may be further organized into hierarchies representing libraries. Therefore, the physical organization of the source code itself serves as an oracle for their expected module classification. Additionally, some of the systems use naming conventions designed to identify components belonging to a module. For instance, the names of all interface functions in a given module have a common unique prefix [9, 8]. Such conventions provide another source of the expected classifications and have already been used as an oracle by some researchers for evaluating SCRTs, as illustrated by the following quote:

> "The two capital letters preceding each of the Fortran procedure names designate the subsystem in which the designers placed the routine." [11, Section III.E.2, page 755]

The scattered-source systems, as the name suggests, do not follow any obvious convention either in organizing the source code or in the naming of its components. Nonetheless, these systems do perform useful functions and are extensively deployed. Additionally, the authors/maintainers of these programs are easily accessible and may be utilized as oracles.

3 A framework for the congruence measure

This section establishes a framework for developing our congruence measure. The framework has been influenced by Basili and Rombach's Goal/Question/Metric (GQM) paradigm [1], Fenton's measurement theory based framework for software measures [7], and a presentation by Bollmann [3].

A *measure* is comprised of three components: (1) an empirical relational system consisting of objects and empirical relations, (2) a numerical relational system, and (3) a homomorphic map from the empirical relational system to the numerical relational system. This can be restated as: a *measure* is an assignment of a number to one or more entities such that *certain* inferences about the entities may be made by using the numbers assigned to them instead of using the entities themselves [7]. The framework of a measure consists of: (1) the purpose of the measure (or the goal of defining the measure), (2) the inferences to be made using the measure (or the questions it should help to answer), (3) the attributes of the entity to be measured, and (4) the empirical relations that the measure should preserve. A measure is meaningful only in the context of the framework for which it is defined (e.g., it makes no sense to say that the temperature is two feet). In the rest of this section we describe the framework for our congruence measure.

The primary purpose of our congruence measure is to provide a capability to quantitatively compare SCs recovered by SCRTs. Our congruence measure quantifies the degree to which components grouped together in one SC are also grouped together in another SC.

Definition: Two components are said to be *differently placed* in two SCs if they are in the same subsystem in one SC but in different subsystems in the other SC [20].

To complete the framework, we now describe the empirical relations that the congruence measure should preserve. The congruence measure should assign a numeric value to a pair of SCs such that it quantifies how similar the two SCs are, and a higher numeric value should denote a greater similarity of the two SCs. Thus, our measure should satisfy the following relations:

R1. The numeric value should be normalized, i.e., be in the range 0 to 1, inclusive, so that two congruence values can be meaningfully compared.

R2. The value assigned should be proportional to the number of components common to both SCs, i.e., if the number of components classified by both of the SCs is low, their congruence should also be low.

R3. The value assigned should be inversely proportional to the number of *common* components that are differently placed, i.e., a high number of pairs of differently placed components should yield low congruence.

The use of the term "proportional" in these empirical relations makes them approximations. The conditions when the measure assigns extreme values, i.e., 0 and 1, will help to illuminate the intuition behind the measure:

- The value 1 should be assigned to a pair of SCs if and only if the two SCs are completely congruent, i.e., the SCs classify the same set of components *and* no pair of components are differently placed in the two SCs.
- The value 0 should be assigned to a pair of SCs if and only if they have nothing in common, which means *either* the two SCs have at most one component in common *or* all pairs of components common to the two SCs are *differently placed.*

Relations R2 and R3 conflict when the number of components common in two SCs is high and all pairs of components are differently placed. However, the fact that these empirical relations are approximate and have a potential conflict does not lessen their utility since, as observed by Fenton, ". . . the empirical relations which necessarily precede objective measurement are normally established initially in approximate form by subjective measures." [7, page 19]. "Once we have identified an attribute and a means of measuring it, we begin to accumulate data. Analyzing the results of this process leads to the clarification and reevaluation of the attribute. This in turn leads to improvements in the accuracy of measurement and an improved scale." [7, page 18]. We expect our measure to evolve also, but we believe this to be an important start.

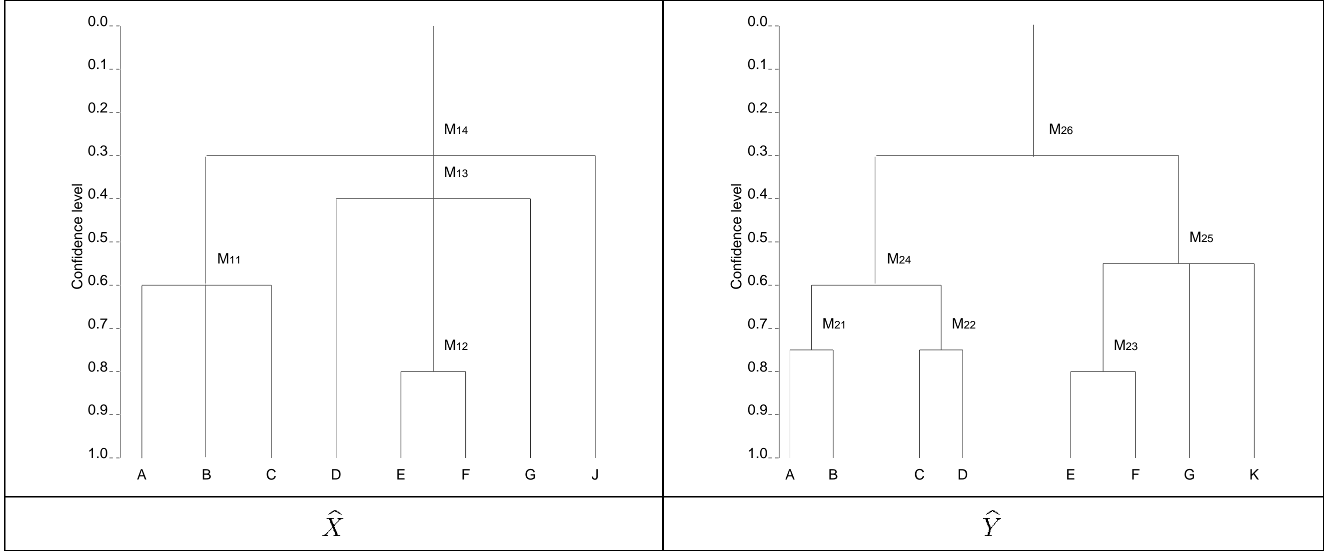

Figure 1. Example dendrogram HSCs.

4 Measure of congruence between SCs

We now present our measure of congruence between SCs. The section is organized as follows. Section 4.1 summarizes the structure of the type of SCs compared by our congruence measure. In Section 4.2 a notion of distance between pairs of components within a classification is developed. These pairwise distances are used in Section 4.3 to compute the distance between two SCs, taking into account only the components classified by both of the SCs. This distance measure satisfies the empirical relation R1 and the *inverse* of relation R3 in Section 3. Finally, in Section 4.4 a measure called the intersection ratio is defined to satisfy relations R1 and R2. The congruence measure is then defined as a function of the intersection ratio and the complement of the distance measure to satisfy empirical relations R1, R2, and R3.

4.1 Structure of SCs

A SC creates a set of subsystems where each subsystem is a set of program components. We restrict our measure to hierarchical subsystem classifications (HSC). A HSC is a classification that may be represented as a tree with intermediate nodes interpreted as representing subsystems and with leaf nodes interpreted as representing program components. The subsystem associated with an intermediate node consists of all of the components at the leaf of the subtree rooted at that node. A HSC has the property that:

> A program component corresponding to a leaf node belongs to the subsystems associated with all of its ancestors.

and as corollaries:

- The subsystem corresponding to the root node contains all of the program components classified by the HSC.
- If two program components belong to a subsystem, then they belong to every larger subsystem containing either of them (since the size of the subsystems increases monotonically as a HSC tree is ascended).

Although several modular SCRTs generate HSCs, the hierarchy of a HSC does not necessarily correspond to a module hierarchy (such as that due to inheritance). On the contrary, a HSC actually gives not one, but several module subsystem classifications with a relative confidence level assigned to each subsystem. A lower confidence level is associated with a subsystem corresponding to a node closer to the root node. Assume that the difference between the confidence levels of any pair of parent and child intermediate nodes is constant, say 1, and that the confidence level of the root node is 0, then the confidence level of each intermediate node corresponds to the distance of that node from the root, i.e., its depth. We refer to such HSCs as simple HSCs.

There are some HSCs in which the difference between the confidence levels of pairs of parent and child intermediate nodes is not constant. Such HSCs are commonly called *dendrograms* [13]. In a dendrogram, a numeric confidence level is assigned to each subsystem, such that the confidence level monotonically increases from a child node to a parent node. A simple HSC may be considered as a special type of dendrogram by using the depth of an intermediate node as its confidence level. Henceforth, we will assume that all HSCs are dendrograms.

Figure 1 presents a diagrammatic representation of two dendrogram HSCs referred to as $\widehat{X}$ and $\widehat{Y}$. The first HSC classifies the components $A..G$ and J, while the second one classifies the components $A..G$ and K. The y-axis in each of these diagrams shows the confidence level for the respective classification. The length of segments

M_i	$L(M_i)$	Subsystem at M_i	M_i	$L(M_i)$	Subsystem at M_i
M_{11}	0.6	$\{A, B, C\}$	M_{21}	0.75	$\{A, B\}$
M_{12}	0.8	$\{E, F\}$	M_{22}	0.75	$\{C, D\}$
M_{13}	0.4	$\{D, E, F, G\}$	M_{23}	0.8	$\{E, F\}$
M_{14}	0.3	$\{A, B, C, D, E, F, G, J\}$	M_{24}	0.6	$\{A, B, C, D\}$
			M_{25}	0.55	$\{E, F, G, K\}$
			M_{26}	0.3	$\{A, B, C, D, E, F, G, K\}$
	$\widehat{X}$			$\widehat{Y}$	

Table 2. Subsytems corresponding to the intermediate nodes of the HSCs in Figure 1.

PDM $\widehat{X}$

	A	B	C	D	E	F	G	J
A	0							
B	0.6	0						
C	0.6	0.6	0					
D	0.3	0.3	0.3	0				
E	0.3	0.3	0.3	0.4	0			
F	0.3	0.3	0.3	0.4	0.8	0		
G	0.3	0.3	0.3	0.4	0.4	0.4	0	
J	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0

PDM $\widehat{Y}$

	A	B	C	D	E	F	G	K
A	0							
B	0.75	0						
C	0.6	0.6	0					
D	0.6	0.6	0.75	0				
E	0.3	0.3	0.3	0.3	0			
F	0.3	0.3	0.3	0.3	0.8	0		
G	0.3	0.3	0.3	0.3	0.55	0.55	0	
K	0.3	0.3	0.3	0.3	0.55	0.55	0.55	0

Table 3. PDMs for the HSCs of Figure 1 computed using the confidence level of the nearest common ancestor of each component pair.

along this axis between two nodes is proportional to the difference in their confidence levels. Table 2 enumerates the subsystems corresponding to the intermediate nodes of dendrogram HSCs $\widehat{X}$ and $\widehat{Y}$.

4.2 Distance between components in a HSC

A pair of components classified by a HSC are never differently placed since they are always placed together in the root subsystem. The boolean nature of the property "differently placed" makes it too weak to compare HSCs. Hence, it is common to introduce another property, the distance between two components in an HSC, also known as *pairwise distance*.

Two pairwise distance measures commonly used in the literature are as follows. The distance between two components p_1 and p_2 in a HSC that classifies both is: (1) the length of the shortest path between p_1 and p_2, ignoring the direction of the edges in the HSC [13], or (2) the confidence level of the nearest common ancestor of p_1 and p_2 in the HSC [6]. These are but two of the many pairwise distance measures that have been proposed [5, 19, 26].

To define our congruence measure for a HSC, it is sufficient to have just the pairwise distances between each pair of components in the HSC, for the distance measure of choice. This information can be represented as a matrix, called a PDM. A PDM is assumed to be indexed by the component names. Let $\mathcal{P}$ represent the universe of all components of all software systems, and let $\mathcal{R}_+$ represent the domain of all non-negative real numbers. A PDM d has the signature $d : \mathcal{P} \times \mathcal{P} \mapsto \mathcal{R}_+$. We use the symbol PDM to denote the domain of all pairwise distance matrices.

Table 3 presents the PDMs for the HSCs $\widehat{X}$ and $\widehat{Y}$ of Figure 1. The PDMs are created using the second pairwise distance measure, i.e., the confidence level of the nearest common ancestor of pairs of components.

4.3 Measure of the difference between HSCs

The difference between HSCs may be measured in terms of their corresponding pairwise distance matrices (PDMs) [13]. Previous works in numerical taxonomy have defined such measures for comparing HSCs that classify exactly identical sets of components, typically the entire universe. HSCs produced by SCRTs do not satisfy this constraint. In the case of SCRTs, even if two techniques classify the same types of program components, they may not agree upon the specific components they classify. This is because a technique may not classify all of the program components of a particular type, usually leaving

out components that do not participate in the interconnection relation(s) it uses [14]. A PDM is, therefore, a partial function, i.e., it does not map every possible pair of components to a real number in that it only gives the pairwise distances between components classified by the corresponding HSC.

Definition: Let $E : PDM \rightarrow 2^{\mathcal{P}}$ be the set of all components in the HSC represented by a PDM. That is, $E(d)$ gives the set of components in the HSC for which d is a PDM.

For example, for $PDM\ \widehat{X}$ and $PDM\ \widehat{Y}$ of Table 3, $E(PDM\ \widehat{X}) = \{A, B, C, D, E, F, G, J\}$ and $E(PDM\ \widehat{Y}) = \{A, B, C, D, E, F, G, K\}$.

Let PDMs d_1 and d_2 represent two HSCs of the same program recovered using different SCRTs. Then $E(d_1) \cap E(d_2)$ gives the set of components classified by both d_1 and d_2. Thus, $E(PDM\ \widehat{X}) \cap E(PDM\ \widehat{Y}) = \{A, B, C, D, E, F, G\}$.

We adapt the previously defined measures of difference between PDMs by constraining them to the domain over which both functions of the PDMs are defined. The adapted measures of difference are stated below.

Definition: Let $\Delta : PDM \times PDM \rightarrow \mathcal{R}_{01}$ be a function that gives the *normalized* "difference" between two PDMs:

- $\Delta_1(d_1, d_2) = \max\{|d_1(z_1, z_2) - d_2(z_1, z_2)| : z_1, z_2 \in Z\}$
- $\Delta_2(d_1, d_2) = \dfrac{\sqrt{\sum_{z_1,z_2 \in Z} [d_1(z_1,z_2) - d_2(z_1,z_2)]^2}}{\sqrt{\frac{1}{2}|Z|(|Z|-1)}}$
- $\Delta_3(d_1, d_2) = \dfrac{\sum_{z_1,z_2 \in Z} |d_1(z_1,z_2) - d_2(z_1,z_2)|}{\frac{1}{2}|Z|(|Z|-1)}$

where d_1 and d_2 are PDMs and Z is the set of components classified by both of the PDMs, i.e., $Z = E(d_1) \cap E(d_2)$. These are, of course, but three of the many proposed pairwise distance measures between PDMs that have been proposed in the literature [5, 19, 24, 26].

The matrix in Table 4 enumerates the intermediate steps for computing $\Delta_1(PDM\ \widehat{X}, PDM\ \widehat{Y})$. A cell in this matrix represents the absolute difference of the pairwise distances between a pair of components in HSCs $\widehat{X}$ and $\widehat{Y}$. Notice that the components J and K do not appear in the calculation since each of them is classified by only one of the HSCs, not both. $\Delta_1(PDM\ \widehat{X}, PDM\ \widehat{Y})$ is 0.45, the maximum of the absolute differences between all pairwise distances.

4.4 Our congruence measure for HSCs

Definition: The *intersection ratio*, $I : PDM \times PDM \rightarrow \mathcal{R}_+$, of PDMs d_1 and d_2 is:

$$I(d_1, d_2) = \frac{|E(d_1) \cap E(d_2)|}{|E(d_1) \cup E(d_2)|}$$

A	0						
B	0.15	0					
C	0	0	0				
D	0.3	0.3	**0.45**	0			
E	0	0	0	0.1	0		
F	0	0	0	0.1	0	0	
G	0	0	0	0.1	0.15	0.2	0
	A	B	C	D	E	F	G

$$\Delta_1\left(PDM\ \widehat{X}, PDM\ \widehat{Y}\right) = max\left\{|PDM\ \widehat{X} - PDM\ \widehat{Y}|\right\} = 0.45$$

Table 4. Calculation of $\Delta_1(PDM\ \widehat{X}, PDM\ \widehat{Y})$ for PDMs of Table 3. Each cell represents the absolute difference of the pairwise distances between two components in $PDM\ \widehat{X}$ and $PDM\ \widehat{Y}$.

The intersection ratio is 1 when both of the recovered HSCs organize the same set of components and it is 0 if they organize entirely different sets of components. Therefore, when comparing two HSCs, a high intersection ratio is “good” and a low intersection ratio is “bad,” which satisfies our framework’s relation R2. For our running example of $PDM\ \widehat{X}$ and $PDM\ \widehat{Y}$ of Table 3, $I\left(\widehat{X}, \widehat{Y}\right) = \frac{7}{9} = 0.78$. The congruence of two HSCs can now be measured as follows.

Definition: The *measure of congruence*, $\mu : PDM \times PDM \rightarrow \mathcal{R}_{01}$, of PDMs d_1 and d_2 is given by:

$$\mu(d_1, d_2) = I(d_1, d_2) \times (1 - \Delta(d_1, d_2))$$

Generally speaking, one may choose any of the Δ functions given in Section 4.3 and compute the PDM using any of the techniques suggested in Section 4.2.

Thus, $\mu(PDM\ \widehat{X}, PDM\ \widehat{Y}) = (0.78) \times (1 - 0.45) = 0.429$ for our example.

5 Conclusions

The problem of recovering the modular subsystem classification of legacy systems is an important issue in the reverse engineering of software. Several techniques have been proposed for recovering such subsystem classifications [14]. There is now a considerable collection of techniques, and the time is now right to experimentally evaluate them.

To evaluate a subsystem classification recovery technique, one must determine the similarity, or congruence, of a recovered classification with an expected classification. This requires two things: (1) the existence of either a set of data for which the expected classifications are known or an *oracle*—a mechanism different from the program itself—that can provide the expected classification, and (2) a *comparator*—a mechanism to compare the output of a program and the expected output—to quanti-

tatively compare the recovered classification with the expected classification. This paper has addressed both requirements. We have proposed a set of widely used and easily accessible as benchmarks for subsystem classification recovery techniques recovering modular subsystem classifications for programs written in the C programming language. We have also presented a measure to quantitatively determine the congruence between two hierarchical subsystem classifications.

Our congruence measure assigns a number in the range of $[0..1]$ to a pair of hierarchical subsystem classifications. The higher the congruence measure, the greater the similarity between the subsystem classifications. Two subsystem classifications are completely congruent if their congruence measure is 1 and completely incongruent if their congruence measure is 0.

Evaluating a subsystem classification recovery technique requires comparing a recovered classification with an expected classification. The effectiveness of a technique is determined by the congruence of its recovered subsystem classifications with the corresponding expected subsystem classifications. This seems to create a chicken-and-egg problem in that, if we know the subsystem classification of a software system, then we do not need to recover it. The problem is, however, superficial, because our congruence measure is intended for use in controlled experiments specifically designed to evaluate how well a technique performs, to compare two techniques, or to identify other causal factors affecting the recovered subsystem classification. Under such controlled conditions, it is quite appropriate to know the expected subsystem classification. The actual design of these experiments is beyond the scope of this paper.

Acknowledgments

This work was partially supported by grants from the Louisiana Board of Regents and from the Army Research Office. The authors wish to thank J. A. Grant, A. Bhatnagar, and V. V. Raghavan for their careful reading and helpful comments in the development of this paper. We also wish to thank the anonymous reviewers whose comments helped us to improve this paper.

References

[1] V. R. Basili and H. D. Rombach. The TAME project: Towards improvement-orientated software environments. *IEEE Transactions on Software Engineering*, 14(6):758–773, 1988.

[2] B. W. Boehm, T. E. Gray, and T. Seewaldt. Prototyping versus specifying: a multiproject experiment. *IEEE Trans. Softw. Eng.*, SE–10(3):290–302, May 1984.

[3] P. Bollman-Sdorra. Basics of measurement. Oral Presentation, March 1995. CACS Colloquia Series.

[4] S. C. Choi and W. Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, 7(1):66–71, Jan. 1990.

[5] W. H. E. Day. The complexity of computing metric distances between paritions. Technical Report 7901, Department of Computer Science, Memorial University of Newfoundland, St. John's, Newfoundland, Canada, September 1979.

[6] J. S. Farris. A successive approximation approach to character weighting. *Systematic Zoology*, 18:374–385, 1979.

[7] N. E. Fenton. *Software Metrics: A Rigorous Approach*. Chapman & Hall, Van Nostrand Reinhold, 1991.

[8] J. V. Guttag and J. J. Horning. Introduction to LCL, a Larch/C interface language. Technical Report SRC 74, Digital Equipment Corporation, July 1991.

[9] J. V. Guttag, J. J. Horning, and A. Modet. Report on the Larch shared language: Version 2.3. Technical Report SRC 58, Digital Equipment Corporation, Apr. 1990.

[10] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego*, pages 133–145, 1988.

[11] D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. *IEEE Trans. Softw. Eng.*, pages 749–757, Aug. 1985.

[12] P. Jalote. *An Integrated Approach to Software Engineering*. Springer-Verlag, New York, NY, 1991.

[13] N. Jardine and R. Sibson. *Mathematical Taxonomy*. John Wiley and Sons, Ltd., New York, 1971.

[14] A. Lakhotia. A unified framework for expressing architecture recovery techniques. *Journal of Systems and Software*, 1995. (to appear).

[15] P. E. Livadas and T. Johnson. A new approach to finding objects. *Journal of Software Maintenance*, 6(4), Sept. 1994.

[16] H. A. Müller and J. S. Uhl. Composing subsystem structures using (K,2)–partite graphs. *Proceedings of the Conference on Software Maintenance*, pages 12–19, Nov. 1990.

[17] R. M. Ogando, S. S. Yau, S. S. Liu, and N. Wilde. An object finder for program structure understanding in software maintenance. *Journal of Software Maintenance Research and Practice*, 6(5), Sep-Oct 1994.

[18] C. Ong and W. T. Tsai. Class and object extraction from imperative code. *J. Object Oriented Programming*, pages 58–68, Mar–Apr 1993.

[19] V. V. Raghavan and M. Y. L. Ip. Techniques for measuring the stability of clustering: a comparative study. In *Research and Development in Information Retrieval, Lecture Notes in Computer Science 146*, pages 200–237. Springer Verlag, 1983.

[20] W. M. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, 66(336):846–850, December 1971.

[21] S. P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66, July 1990.

[22] T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Contructing Language-Based Editor*. Springer-Verlag, New York, NY, 1988.

[23] T. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual*. Springer-Verlag, New York, NY, third edition, 1988.

[24] F. J. Rohlf. Methods of comparing classifications. *Annual Review of Ecology and Systematics*, 5:101–113, 1974.

[25] R. Schwanke. An intelligent tool for reengineering software modularity. In *Proc. 13th International Conference on Software Engineering*, 1991.

[26] R. R. Sokal and F. J. Rohlf. The comparison of dendrograms by objective methods. *TAXON*, XI(2):33–40, 1962.