# Architecture recovery techniques: a unified view and a measure of their goodness

**Arun Lakhotia**

Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504
arun@cacs.usl.edu
(318) 231-6766, Fax: -5791

***Draft: Please do not quote or circulate. Thanks.***

## Abstract

A partitioning of software system into modules is often termed as its *architecture*. Each partition (or module) consists of a set of functions, procedures, global variables, and/or type declarations. To help maintain legacy systems or systems with inadequate documentation, several approaches have been proposed for recovering their architectures from their source code. This paper makes three contributions. First, it develops a scheme to classify these techniques. Second, it surveys the various architecture recovery techniques and presents them using a unified and consistent terminology. Third, it presents a measure to evaluate 'how well a recovered architecture matches an expected (or actual) architecture'. It is envisioned that this measure may be used in carefully controlled experiments to discern the contexts of applicability of various techniques and to pave the way for their improvement. Results from an experiment to evaluate the performance of two architecture recovery techniques suggested by Hutchens and Basili are presented.

## 1 Introduction

It has been widely acknowledged that software maintenance consumes 60–80% of a software system's total cost [LS80]. The activities performed during maintenance may broadly be classified as: a) understand how the software functions, b) design and implement a change, and c) validate the change [Par86]. The working of a program is typically understood by reading its source code and documentation. While reading the documentation is most helpful, it is very rare to find a documented software system. And when such a document is available it is very often inconsistent since it has not been kept up-to-date with the modifications done to the software system [LB85]. The only definitive document of a program, therefore, is the program itself.

Research efforts in design recovery are aimed at extracting the design of a software system from its source code, where the term "design" loosely means any higher level abstraction of a software system beyond the source code itself [Chi90]. One such design is *architectural design* - a partitioning of structural elements (such as functions, procedures, variables) of a software system into groups of related components. This is at a higher level of abstraction than a design describing the algorithms or data structures used in a system.

Several researchers have proposed automated or semi-automated techniques to create groups of related program symbols with the implicit or explicit objective of recovering the modular decomposition[*] of the program [BE81, CS90, HB85, MMM90, MBK91, LW90, PCB92, Sch91, SB91][†]. It is not very obvious how these Architecture

[*] The term "module" is used in this paper in the sense used by Parnas [Par76] in the context of abstract data typing and information hiding.

[†] Some researchers have proposed architecture recovery techniques that are not automatable because of their dependence on analyses performed by human [JL91, SS94]. These are not studied in this paper.

Recovery Techniques (ARTs) compare with each other since the techniques have been expressed using different sets of terms and symbols. It is also not clear how successful the techniques are, individually or in comparison to other techniques, in recovering the actual architecture. This is because of the absence of any measure to evaluating how "close" an architecture recovered by a method is a) compared to an expected architecture and b) compared to architectures recovered by other techniques.

This paper provides a framework for comparing ARTs. It a) develops a scheme to classify ARTs, b) develops a unifying view of the ARTs by rephrasing each technique using a common set of symbols and terms, and c) develops a metric to quantify how "close" a recovered architecture is to an expected architecture.

The rest of the paper is organized in eight sections, excluding references. Section 2 gives our classification scheme for ARTs. Section 3 summarizes methods for numeric cluster analysis and measures for comparing clusters found by these methods. Several ARTs are based on cluster analysis. Section 4 introduces our terminology to express the ARTs. Section 5 surveys and summarizes several ARTs. It should be emphasized that in some cases, our presentation may be significantly different from that presented in the original works. This is because we have attempted to present all the techniques using the same set of symbols and concepts. Section 6 gives our measure for comparing architectures. It is based on the measures presented in the previous section. Section 7 summarizes the results of an experiment we conducted to evaluate the correctness of the ARTs suggested by Hutchens & Basili [HB85]. Section 8 contains our concluding remarks. It also outlines an experiment on how our measure may be used in carefully controlled experiments to discern the contexts of applicability of various techniques and to pave the way for their improvement.

# 2 A scheme for classifying ARTs

An architecture of a software system is analogous to classification hierarchies, called *clusters*, used by statisticians [Eve74] (see Section 3). Recovering a software's architecture corresponds to performing cluster analysis. The term *cluster analysis* broadly refers to any method of grouping a set of objects such that objects in the same group are in "some sense" more similar to each other than to those in different groups. Clustering has been used in the life-sciences to classify organisms since the early 1700s. It is now commonly used in the social sciences, in information retrieval, and in pattern recognition to organize and understand large amounts of data. Its first use in organizing design information dates back to the 1960s when the British architect Christopher Alexander used it to organize architectural constraints for designing cities [Ale64]. Alexander used cluster analysis to generate a "program" from design constraints. To him a "program" was a sequence of instructions to the architect. Though a mere play of words, it is interesting to observe that while Alexander used cluster analysis to get a program from a design, the ARTs use it to extract a design from a program.

Clustering techniques may be classified based on the properties of the resulting grouping or the information used in creating these groups, as follows:

- If the pairwise intersection of the resulting groups is empty then the technique is *partitioning*, else it is *overlapping* (or *non-partitioning*).
- If the technique creates *levels* of groups it is *stratified*. If there is only one level of grouping it is *non-stratified* (or *flat*). If the stratified groups form a tree then it is *hierarchic*.
- A technique may be classified as *conceptual*, *graph theoretic*, or *numerical* depending on the type of information and computation procedures used. When values of object attributes are measured on an ordinal scale the method is classified as conceptual clustering [MS83]. If the technique uses graph-theoretic computations, it is termed graph-theoretic. However, if the attribute values are measured on a ratio or an interval scale and numeric computations are used, the technique is called numeric.

Numeric, hierarchic, and partitioned clustering techniques have been in common use for a long time and hence are the most well studied techniques. There is a rigorous generalized framework that may be used to describe properties of numerical methods. In contrast, the conceptual and graph-theoretic methods have not been used as frequently and hence do not have a generalized framework. They, therefore, must be studied individually.

The architecture of a software system recovered using ARTs, to some, may appear to be of questionable value. Most of the techniques are heuristic. They use cross reference information from a program to identify its modules. Critics may rightly cite several scenarios where such cross-reference information may not be sufficient to extract a software's architecture. To argue in favor of the ARTs, it is worth observing that maintenance programmers find the cross-reference data as one of the most important source of information [Par86]. The alternative is to use information from control and data flow analyses [Hec77, MJ81] . But such information would be insufficient since the notion of information hiding is defined on the basis of scope and operations on symbols [Par76], not the information they generate and propagate. Besides, 75-80% of the code currently under maintenance is in languages such as COBOL, FORTRAN, CMS-2, and JOVIAL, languages designed when structured programming was still a buzzword [Sch87]. For these languages data and control flow analyses would yield very imprecise results. Thus, cross-reference data may provide information as good as other static analyses. Ideally, the two types of information may be used in conjunction with flow analysis, for instance Hutchens & Basili [HB85] use cross-reference relationships defined using information obtained from a program's flow analysis.

# 3 Basics of numeric cluster analysis

This section summarizes from the literature the theoretical framework of numeric, hierarchic, partitioning, clustering algorithms (NHPC) and measures for comparing clusters. Most of the work is summarized from Jardine & Sibson [JS71].

## 3.1 Cluster analysis: dissimilarity matrices to dendrograms

**Definition:** Let $P$ denote a set of objects with $p$ elements. A *relation* on $P$ is a subset of $P \times P$. Let $\Xi(P)$ denote the set of all equivalence (symmetric, reflexive, and transitive) relations on $P$. Let $\mathcal{R}_+$ denote the set of real numbers $\geq 0$ and $\mathcal{R}_{01}$ the set of real numbers between $[0, 1]$.

An NHPC method creates a *dendrogram* from some *dissimilarity coefficients*.

**Definition:** ([JS71]). A *dissimilarity coefficient*, also referred to as a dissimilarity matrix or DC, on a set $P$ is a function, $d : P \times P \rightarrow \mathcal{R}_+$, satisfying the following conditions:

DC1. $d(A, A) = 0, \forall A \in P$
DC2. $d(A, B) = d(B, A), \forall A, B \in P$

**Definition:** ([JS71]). A *dendrogram* is a function $c : \mathcal{R}_+ \rightarrow \Xi(P)$ satisfying the following conditions:

D1. $0 \leq h \leq h' \Rightarrow c(h) \subseteq c(h')$
D2. $c(h) = P \times P$ for large enough $h$.
D3. Given $h$, $\exists \delta > 0$ such that $c(h + \delta) = c(h)$

In other words, a dendrogram maps non-negative real numbers, called *levels*, to equivalence relations, or clusters, on $P$ such that 1) the cluster at a level is completely contained in a cluster at a higher level and 2) at some level $h$, $c(h)$ is eventually $P \times P$. The third condition establishes uniqueness in cases where $c$ is discontinuous.

A dendrogram is usually represented as a tree with numeric levels associated to its branches; see Figure 1 for an example. The abscissa of a dendrogram has no specific meaning but the ordinate represents levels. If a line is drawn parallel to the abscissa then the leaves of each tree rooted at the branches it cuts represent the partitions due to the equivalence class at that level. The set of all these equivalence classes (or partitions) is represented by a dendrogram.

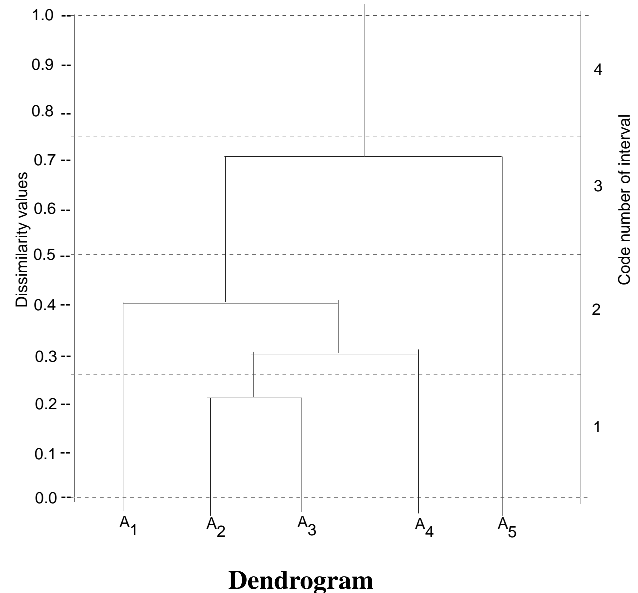
**Definition:** Let $UC(P)$ and $UG(P)$ represent the sets of all DCs and dendrograms, respectively, over the set $P$. We will ignore the parameter $P$ and simply use $UC$ and $UG$; the parameter will be obvious from the context of usage.

**Definition**: A NHPC method $H$ is a mapping $H : UC \rightarrow UG^*$.

* An analogous hierarchical clustering method can be defined using *similarity coefficients* and dendrograms with decreasing level numbers. There is a simple transformation from a similarity based clustering method to an analogue dissimilarity based clustering method. Most of our discussion is from the point of view of dissimilarity based method. The analogue for similarity based method is obvious and, for the sake of simplicity, is omitted.

| | | | | | |
|---|---|---|---|---|---|
| $A_1$ | 0 | | | | |
| $A_2$ | 0.4 | 0 | | | |
| $A_3$ | 0.6 | 0.2 | 0 | | |
| $A_4$ | 0.9 | 0.5 | 0.3 | 0 | |
| $A_5$ | 0.9 | 0.8 | 0.8 | 0.7 | 0 |
| | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ |

**Dissimilarity matrix**




**Dendrogram**

Figure 1 A sample dissimilarity matrix and the corresponding dendrogram resulting from single-link HAC. Since the matrix is symmetric, only the lower triangle and the diagonal are shown. A dendrogram is a hierarchy of set of equivalence relations (i.e. partitions). It may be represented as a tree. The partitions at any dissimilarity value may be determined by drawing a line perpendicular to the dissimilarity values axis. Each branch of the tree cut by the line represents a partition consisting of elements in the subtree rooted at that branch. Elements in a partition at a lower dissimilarity value are more likely to be similar. At the highest level of dissimilarity value all elements are in the one partition. Instead of the possibly infinite number of levels (since range is $\mathcal{R}_+$) one may divide the levels into some fixed levels of intervals. The axis on the right represents one such interval assignment.

```
input: a DC 'd'on a set 'P'
output: dendrogram
put each element of 'P' in a group by itself
while 'd' has more than one element do
   s1: identify the two most similar groups
   s2: combine the elements and create a new 'd'
end-while
```

Figure 2 Outline of HAC algorithms. An HAC algorithm may be generated by choosing a strategy for identifying the two most similar elements (step *s1*) and a strategy for "combining" these elements to create a new dissimilarity matrix (step *s2*).

A class of NHPC algorithms called the *hierarchical agglomerative clustering*, or HAC, have the outline given in Figure 2. An HAC algorithm may be generated by choosing a strategy for identifying the two most similar elements (step *s1*) and a strategy for "combining" these elements to create a new dissimilarity matrix (step *s2*). The pair of elements with the smallest coefficient in a dissimilarity matrix are usually considered as the most similar elements. Analogously, if a similarity matrix is used, the pair with the highest coefficient would be most similar.

There are four popular strategies for "combining" similar elements, called *single-link*, *complete-link*, *weighted-average-link*, and *unweighted-average-link*. The single-link algorithms take the smallest dissimilarity between the pair of similar elements as the coefficient for the new element representing their agglomeration. The complete-link algorithms take the largest such dissimilarity coefficient and the other methods use weighted and unweighted averages.

The dendrogram in Figure 1 is the result of applying the single-link HAC on the dissimilarity matrix in the same figure. In this example, $A_2$ and $A_3$ combine at level $0.2$, $A_4$ combines with $A_2$ and $A_3$ at level $0.3$, $A_1$ combines with $A_2$, $A_3$, and $A_4$ at level $0.4$. Finally, all the objects form a single cluster at level $0.7$.

|  |  |  |  |  |  |
|---|---|---|---|---|---|
| $A_1$ | 0 |  |  |  |  |
| $A_2$ | 0.4 | 0 |  |  |  |
| $A_3$ | 0.4 | 0.2 | 0 |  |  |
| $A_4$ | 0.4 | 0.3 | 0.3 | 0 |  |
| $A_5$ | 0.3 | 0.3 | 0.3 | 0.3 | 0 |
|  | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ |

**Equivalent to the dendrogram**

|  |  |  |  |  |  |
|---|---|---|---|---|---|
| $A_1$ | 0 |  |  |  |  |
| $A_2$ | 2 | 0 |  |  |  |
| $A_3$ | 2 | 1 | 0 |  |  |
| $A_4$ | 2 | 2 | 2 | 0 |  |
| $A_5$ | 3 | 3 | 3 | 3 | 0 |
|  | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ |

**Cophonetic values**

|  |  |  |  |  |  |
|---|---|---|---|---|---|
| $A_1$ | 0 |  |  |  |  |
| $A_2$ | 4 | 0 |  |  |  |
| $A_3$ | 4 | 2 | 0 |  |  |
| $A_4$ | 3 | 3 | 3 | 0 |  |
| $A_5$ | 3 | 5 | 5 | 4 | 0 |
|  | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ |

**Cladistic difference**

Figure 3 Dissimilarity matrix created from the dendrogram in Figure 1. Since it is not convenient to compared dendrograms, for the sake of comparison, it is preferred to transform dendrograms into dissimilarity matrices. The first matrix contains the lowest dissimilarity value at which two elements are placed in the same partition. The matrix is equivalent to the dendrogram in that one can be derived from the other. The second matrix contains the code number of the interval (given by the right axis in the dendrogram) for the dissimilarity value of the corresponding entry of the first matrix. If the edges of the tree representation of a dendrogram are considered to be undirected then the third matrix contains the length of the path from one element to another.

## 3.2 Transforming dendrograms to dissimilarity matrices

The NHPC based ARTs create clusters that may be represented as hierarchies. To define a metric for evaluating how well a recovered architecture matches an expected architecture such hierarchies need to be compared. Since it is very unwieldy to define operations such as comparison of hierarchies, it is customary to transform a dendrogram back into a dissimilarity matrix. This requires defining a function $U : UC \rightarrow UG$. There are several ways to do it. Figure 3 gives three dissimilarity matrices derived from the dendrogram in Figure 1. The first matrix in this figure represents the dendrogram without any loss of information, i.e. the same dendrogram may be recreated from this matrix. The following function defines this transformation*.

**Definition:** $U_1(c) = (\lambda A, B) \cdot (\min \{h \mid (A, B) \in c(h)\})$.

The mapping $U_1(c)$ maps an pair of elements to the smallest level at which they are in the same cluster. Condition DC3, above, guarantees that $U_1$ is 1-1 from the set of dendrograms to some subset of the set of DC's on P. The dendrogram $c$ associated to $U_1(c)$ is given by:

$c(h) = \{(A, B) \mid U_1(c)(A, B) \leq h\}$

Thus, in our subsequent discussion on comparison of dendrograms we use $U_1(c)$ instead of $c$.

**Notation:** We treat function application as left associative. That is, $f(x)(y) \equiv (f(x))(y)$.

There are other ways to create a dissimilarity coefficient from a dendrogram. Sokal and Rohlf [SR62] give a variation of $U_1$ by dividing the levels of the dendrogram into intervals and using the interval number instead of the level. Suppose that the dissimilarity values range between 0 and 1. The range may be split into $N$ intervals of equal size and a code number from 1 to $N$ may be associated to each interval as follows. The interval $(i - 1/N, i/N]$, for $1 \leq i \leq N$, is assigned the code $N - i + 1$. A dissimilarity value may be mapped to the code number of the interval in which it is contained. The following function: $K : R_{01} \rightarrow 1..N$ does this.

**Definition:** $K(j) = N - i + 1$ such that $j \in (i - 1/N, i/N]$, for $1 \leq i \leq N$.

Sokal and Rohlf defined the *cophonetic value* of a pair of elements as the smallest number interval in which the pair are placed in the same operation. The cophonetic values for all pairs of elements form a dissimilarity matrix. The mapping is defined as follows.

**Definition:** $U_2(c) = (\lambda A, B) \cdot (K(U_1(c)(A, B))$.

The second matrix in Figure 3 gives the cophonetic values with $N = 4$ for the dendrogram in Figure 1.

A significantly different method using the notion of *cladistic difference* is suggested by Farris [Far79].

---

* We use subscripts to distinguish more than one function in a particular class, e.g. $U_1, U_2, \Delta_1, \Delta_2, \ldots$

**Definition:** The *cladistic difference* between two leaves of a dendrogram is the number of edges in the path between them in a tree representation of the dendrogram. We use $U_3$ to denote the mapping from a dendrogram to the dissimilarity coefficients due to cladistic difference between elements.

The third matrix in Figure 3 is created by applying $U_3$ to the dendrogram in Figure 1.

For the sake of mathematical convenience we assume that, henceforth, the range of a dendrogram is $\mathcal{R}_{01}$ and not $\mathcal{R}_+$. There is no loss of generality since this can be achieved by dividing all the levels of a dendrogram by the smallest level at which all elements of $P$ are in the same cluster. This mapping essentially normalizes a dendrogram and makes it independent of scale, a property that eases the comparison of dendrograms.

## 3.3 Measures of difference between clusters

Cluster analysis techniques are statistical. The computation they use to group program elements are essentially heuristic. Once a technique has been designed, one needs to evaluate it. Jardine and Sibson have identified several factors on which the plant taxonomy resulting from cluster analysis may be evaluated [JS71]. Their factors, adapted to the context of software systems, are as follows:

1. *Correctness*: How effective is the technique in recovering "actual" module hierarchy?
2. *Stability*: How sensitive are the recovered clusters to perturbation of data?
3. *Consistency*: How sensitive are the recovered clusters to the order in which data is processed.
4. *Test of independence*: Do the classifications depend upon any other properties of the population? For instance, the classifications may be dependent upon the programming standards used in an organization.
5. *Measurement of dependence*: The extent to which the classifications depend upon other properties of the population (as against simply testing for independence).
6. *Measurement of congruence*: How well do the classifications based on one set of criteria 'fit' those due to another set? For instance, whether a module classification arrived by analyzing comments and documents [MBK91] matches that due to analyzing the bindings between various symbols [HB85].

To evaluate each of these factors requires comparing multiple classifications generated by a method. To enable experimentation the comparison must be objective and computable. There has been a significant amount of work on measures for comparing classifications. They are summarized below. Since comparison of dendrograms is not convenient, as stated earlier, to ease comparison it is typical to map the dendrograms to dissimilarity matrices. The $U$ functions of the previous subsection perform such mappings and are used by the comparison functions stated in the following subsections.

**Definition:** Let $\Delta : DC \times DC \rightarrow \mathcal{R}_{01}$ be a function that gives the *normalized* 'difference' between two DCs. Jardine and Sibson [JS71] give the following functions:

1. $\Delta_1(d_1, d_2) = \max\{|d_1(A,B) - d_2(A,B)| : A, B \in P\}$
2. $\Delta_2(d_1, d_2) = \frac{\sqrt{\sum [d_1(A,B) - d_2(A,B)]^2}}{\sqrt{\frac{1}{2}p(p-1)}}$
3. $\Delta_3(d_1, d_2) = \frac{\sum |d_1(A,B) - d_2(A,B)|}{\frac{1}{2}p(p-1)}$

where $d_1$ and $d_2$ are two DCs and the summations are over the $p(p-1)/2$ two element subsets of $P$.

The difference between two dendrograms can also be measured by their Euclidean distance if they are represented by points on the Euclidean space of $p(p-1)/2$ dimension. The Euclidian distance is not a good measure because it is often hard to establish the necessary triangle inequality for such a Euclidean space.

Sokal and Rohlf [SR62] have suggested using the product moment correlation coefficient between $d_1 = U(c_1)$ and $d_2 = U(c_2)$ to compare two classifications $c_1$ and $c_2$. This coefficient, which they term *cophenetic correlation coefficient*, predicts the extent to which there is a linear relationship between the two DCs and may be used as a measure of their relationship.

Day [Day79] has compiled a set of distance measures between partitions (hierarchical, flat clusters) based on lattice- and graph-theoretic representations. Rohlf [Roh74] and Day [Day79] survey over 15 comparison functions taken from literature. The choice of the appropriate $\Delta$, as is the case with any metric, depends on the application.

Jardine and Sibson [JS71], for instance, find $\Delta_3$ as the preferred metric for comparing animal and plant taxonomies. They argue that $\Delta_1$ is not very sensitive and $\Delta_2$ requires attaching a meaning to squaring of DC values, which may not always be appropriate.

### 3.4 Correctness of a clustering technique

The difference between a dendrogram generated by an ART and an expected dendrogram may be measured by an appropriate $\Delta$ function. The centroid of the differences between a large number of such samples may be taken as a measure of its correctness. The choice of statistic for the centroid will depend on the distribution of the distances; for instance, arithmetic mean may be used if the distribution is normal.

## 4 Terminology for unifying ARTs

This section introduces symbols and terms used in the next section to present various ARTs within a unifying framework.

All the ARTs studied have one thing in common. They take as input information about cross-references between program symbols, such as those extracted by [BHW89, CNR90, Rei90, WCW88]. For most techniques, the set of program symbols is a subset of the names of functions, procedures, types, global variables, and files. Schwanke's technique [Sch91] also considers names of algorithms found in textbooks (which is different from the names of functions implementing them). Maarek et. al.'s technique [MBK91] considers words used in comments and documentation. Based on these relationships they attempt to organize the program symbols in collections representing modules. They differ in:

1. the set of program symbols they use,
2. the type of relations they observe,
3. the processing they do to identify the modules,
4. the type of symbols that are included in a module.

The set of program symbols used by an individual technique may further be divided into two sets: the set of *architecture elements* $\mathcal{P}$ and the set of *resources* $\mathcal{F}$. The difference between the two sets is that while the symbols from both the sets may be used by a technique, the recovered architecture is specified only in terms of the symbols in set $\mathcal{P}$. What constitutes $\mathcal{P}$ and $\mathcal{F}$ differs for each ART. Furthermore, not all elements of $\mathcal{P}$ may appear in the recovered architecture. We classify the subset of $\mathcal{P}$ appearing in the recovered architecture as $P$. In other words, although the symbols $\mathcal{P} \cup \mathcal{F}$ are used by an ART, the architecture is defined only in terms of $P \subseteq \mathcal{P}$.

The relationship between program symbols used by various ARTs may further be classified into two groups:

1. Resource usage diagrams (RUD) and
2. Resource flow diagrams (RFD)

### 4.1 Resource usage diagrams

An RUD is a collection of binary relations of the form $\pi : \mathcal{P} \times \mathcal{F}$. These relations, for instance, $\pi_r$, $\pi_u$, $\pi_c$, etc., defined later, represent relationships such as *refer*, *use*, *called-by*, respectively. A collection of these relations may also be represented as a labelled directed graph whose nodes belong to $\mathcal{P} \cup \mathcal{F}$, edges go from a node in $\mathcal{P}$ to a node $\mathcal{F}$, and the label of the edge corresponds to the specific relation. An RUD, therefore, encodes cross-reference relationships between program symbols.

Following is a list of $\pi$ relations used by the ARTs studied in this paper: The first three relations are between procedures and global variables.

1. $\pi_a$: $\pi_a(p, v)$ means procedure $p$ defines (i.e. assigns to) global variable $v$.
2. $\pi_u$: $\pi_u(p, v)$ means procedure $p$ uses global variable $v$.
3. $\pi_r$: $\pi_r(p, v)$ means procedure $p$ refers to (i.e. either defines or uses) global variable $v$.

4. $\pi_c$: A relation between two procedures. $\pi_c(p_1, p_2)$ means procedure $p_1$ calls procedure $p_2$.
5. $\pi_w$: A relation between programs and pairs of *lexical affinity* words. Two words have a lexical affinity in a document if they appear in it with in $\pm 5$ word distance. $\pi_w(p, ws)$ means the lexical affinity $ws$ appears in the documentation of program $p$.
6. $\pi_i$: A relation between procedures and types. $\pi_i(p, t)$ means procedure $p$ either has a formal parameter of type $t$ or returns a value of type $t$. In other words, type $t$ is used in the interface specification of a procedure $p$.
7. $\pi_t$: A relation between procedures and types. $\pi_t(p, t)$ means type $t$ appears in procedure $p$. $\pi_i \subseteq \pi_t$.
8. $\pi_d$: A relation between files and all types of program elements. $\pi_d(f, s)$ means program element $s$ is declared in file $f$.
9. $\pi_f$: A relation between pairs of files. $\pi_f(f_1, f_2)$ means file $f_1$ includes file $f_2$.

An RUD defined as a labelled directed graph implies that there is at most one edge of a particular type between any two nodes. Some of the ARTs we study use the number of occurences of each relationship. This information may be abstracted using a labelled, directed *multi*graph. To accommodate this, we introduce, for every $\pi$ relation defined above, a mapping $\Pi : \mathcal{P} \times \mathcal{F} \rightarrow \mathcal{N}$, where $\mathcal{N}$ is the set of natural numbers. Thus $\Pi_r(p, v)$ gives the number of times a procedure $p$ refers to a global variable $v$.

Notice that $\pi(x, y) \Leftrightarrow \Pi(x, y) > 0$ and $\neg\pi(x, y) \Leftrightarrow \Pi(x, y) = 0$.

The set of architecture elements $\mathcal{P}$ for an ART is obvious from the discussion. The set of all program elements $\mathcal{P} \cup \mathcal{F}$ used by an ART is evident from the choice of the $\pi$ relations or $\Pi$ functions it uses. We, therefore, do not explicitly state these sets for each technique.

**Notation:** For the sake of convenience, we map the boolean value *true* to $1$ and *false* to $0$. This allows us to use the $\pi$ relations in arithmetic expressions.

**Notation:** A '$-$' is used as a shorthand for an existentially quantified symbol. Two '$-$' symbols in the same expression represent two different existentially quantified symbols. The expression $\xi(-, B, -)$, where $\xi$ is a relation, is a shorthand for $\{z : \exists X, Y \cdot z = \xi(X, B, Y) \wedge \xi(X, B, Y)\}$

## 4.2 Resource flow diagrams

A resource flow diagram is also a labelled directed graph. The vertices of this graph, however, consist only of elements from the set $\mathcal{P}$, the set of architectural elements. The edges of an RFD are labelled by the elements from the set $\mathcal{F}$. The set of labels of an RFD, therefore, depends on the program being analyzed. Contrast this with an RUD whose vertices belonged to $\mathcal{P} \cup \mathcal{F}$ and whose labels were fixed based on the type of relationship considered.

An edge in an RFD from vertex $x$ to vertex $y$ with label $z$ indicates that program element $x$ provides resource $z$ to program element $z$. Stated another way, resource $z$ flows from program element $x$ to program element $y$. An RFD, therefore, is a ternary relation $\psi : \mathcal{P} \times \mathcal{F} \times \mathcal{P}$.

The notion of what "flows" mean may differ on the specific relationship captured. Hutchens and Basili, for instance, identify the following relations:

1. $\psi_f(p_1, v, p_2)$ – a relation established if (a) procedure $p_1$ modifies variable $v$, (b) procedure $p_2$ uses variable $v$, and (c) there is an executable path from procedure $p_1$ to procedure $p_2$ over which variable $v$ is not modified.
2. $\psi_c(p_1, v, p_2)$ – a relation established if (a) procedure $p_1$ modifies variable $v$, (b) procedure $p_2$ uses variable $v$, and (c) there is an executable path from procedure $p_1$ to procedure $p_2$.
3. $\psi_s(p_1, v, p_2)$ – a relation established if procedure $p_1$ modifies variable $v$ and procedure $p_2$ uses it

For relations $\psi_f$ and $\psi_c$ the notion of "flows" depends on the existence of an execution path from one procedure to another. Establishing such relationships may often require static flow analysis [ASU86, Hec77]. In contrast the $\psi_s$ requires information local to individual procedures without concern of existence of execution paths connecting them. Such relations may often be computed from $\pi$ relations. As for instance:

$\psi_s(p_1, v, p_2) = \pi_a(p_1, v) \wedge \pi_u(p_2, v)$.

Table 1 Classification of ARTs based on criteria introduced in Section 2

| *Source* | *Type of computation performed* | *Generates paritions?* | *Generates stratified architectures?* |
|---|---|---|---|
| Belady & Evangelisti, 81 | numeric | yes | no |
| Choi & Scacchi, 90 | graph-theoretic | yes | yes |
| Hutchens & Basili, 85 | numeric | yes | yes |
| Liu & Wilde, 90 | graph-theoretic | no | no |
| Maarek *et. al.*, 91 | numeric | yes | yes |
| Müller & Uhl, 90 | mixed (numeric & graph-theoretic) | no | yes |
| Patel *et. al.*, 92 | numeric | no | no |
| Schwanke, 91 | numeric | yes | yes |
| Selby & Basili, 91 | numeric | yes | yes |

Analogous to $\Pi$, we define a mapping $\Psi : \mathcal{P} \times \mathcal{P} \rightarrow \mathcal{N}$ that counts the number of resources flowing from one element to another in a $\psi$ relation. More precisely:

$$\Psi(a, b) = |\psi(a, -, b)|$$

For every RFD $\psi$ we may also define a RUD $\pi_\psi : \mathcal{P} \times \mathcal{P}$ as follows:

$$\pi_\psi(a, b) = |\Psi(a, -, b)| > 0$$

# 5 A unified view of ARTs

This section gives an overview of various architecture recovery techniques. The overview presents a unified view in that, instead of using terms and symbols from the original work, the terms and symbols introduced in the previous section are used. This allows one to compare the information used, the computation performed, and the output generated by various techniques.

Table 1 classifies ARTs studied in this paper based on the classification scheme developed in Section 2. The first column of this table cites the paper in which the technique was presented. The papers of [HB85, LW90] present multiple ARTs, each of them with the same classification. The second column classifies the techniques on the type of computation they perform, i.e. graph-theoretic, numeric, or conceptual. Notice that none of the ARTs use conceptual clustering. The ARTs of [BE81, HB85, PCB92, MBK91, Sch91, SB91] use numeric computations, those in [CS90, LW90] use graph-theoretic computations. The ART of [MU90] uses both graph-theoretic and numeric computations. The third column classifies each technique on whether they place a program element in one and only one module (partitioned), or whether a program element may be placed in more than one module (non-partitioned). The techniques of [BE81, CS90, HB85, MBK91, Sch91, SB91] create partitions while those of [LW90, MU90, PCB92] do not. The fourth column classifies each technique on whether they organize program elements in levels. All the ARTs, except those of [BE81, LW90, MU90, PCB92], create hierarchic architectures. The architecture generated by ART of [MU90] are stratified but not hierarchical.

## 5.1 Numeric, stratified ARTs

The ARTs of [HB85, MBK91, Sch91, SB91] are numeric and stratified in that they use numeric computations to create not one but several levels of modules. The architectures recovered by these techniques are in fact hierarchical

and the modules are partitioned. All these ARTs use the outline of HAC algorithms, Figure 2, though with some modifications. Hence, to compare these techniques we only need to identify:

1. Which $\pi$ relations or $\Pi$ functions they use?
2. How the dissimilarity of similarity matrix is created?
3. In the HAC loop (Figure 2), what strategy is used to:
   a. select the most similar elements (step *s1*)?
   b. "combine" these elements to create a new similarity or dissimilarity matrix (step *s2*)?

Table 2 gives a comparative analysis of numeric architecture recovery techniques. It represents the information used by these techniques using the relation $\pi : \mathcal{P} \times \mathcal{F}$. Since different techniques use different information their architectures may differ on the structural elements they organize. Due to differences in the computations used different techniques may classify the same program element in different clusters.

#### 5.1.1 ARTs of Hutchens & Basili [HB85]

Hutchens & Basili's ARTs maintain an RUD consisting of $\pi_r$ (which procedure refers to which global variable) and $\pi_d$ (which procedure modifies which global variable) relations. They suggest two methods based on two different methods for computing dissimilarity matrices: *recomputed* and *expected* dissimilarity matrices. The recomputed dissimilarity matrix is computed as follows:

$$diss_1(A,B) = \frac{s(A) + s(B) - 2b(A,B)}{s(A) + s(B) - b(A,B)}$$

where $b$ is the binding matrix which gives the *binding strength* between two procedures and the vector $s$ gives the number of bindings involving a given procedure. A binding is a ternary relation: $e(A,x,B)$ representing that procedure $A$ refers to a global variable $x$ that is modified by procedure $B$. Clearly, $e(A,x,B) = \pi_r(A,x) \wedge \pi_a(B,x)$. The matrix $b$ and the vector $s$ are defined as follows:

$$b(A,B) = |e(A,-,B)| + |e(B,-,A)|$$

$$s(A) = |e(A,-,-)| + |e(-,-,A)|$$

Remember that $e(A,_,B) = \{e(A,x,B) \mid \forall x \cdot e(A,x,B)\}$, and similarly $e(A,_,_)$ and $e(_,_,A)$.

The expected dissimilarity is defined as:

$$diss_2(A,B) = \frac{s(A) + s(B)}{(\dim(b) - 1) \times b(A,B)}$$

where $b$ and $s$ are as defined for recomputed dissimilarity and $dim(b)$ gives the dimension of $b$.

The ARTs of Hutchens & Basili use a modification of HAC. They maintain two matrices: a binding matrix and a dissimilarity matrix. Instead of computing the latter only once, as done in HAC, Hutchens & Basili's ART recomputes it in each iteration. In the HAC loop, the two most similar elements are chosen based on the smallest dissimilarity. Then the binding matrix is recreated by "combining" these similar elements using the single-link method. The dissimilarity matrix is recomputed using the appropriate function (stated above).

The recomputation of dissimilarity matrix at each iteration leads to the possibility that the dissimilarity values over successive iterations decrease, implying that a later iteration may group a pair with lower dissimilarity. The hierarchy of partitions thus created would not always have non-decreasing level numbers. The resulting classification therefore will not be a dendrogram and hence to easy to interpret. To overcome this problem, Hutchens & Basili suggest that whenever the dissimilarity of a newly created cluster is smaller than that of the previous iteration, the new cluster be merged with the previous cluster. This may be done by assigning the new cluster the same dissimilarity level as the previous cluster.

**Table 2 Overview of various *numeric* architecture recovery techniques.**

| Design recovery method | $\mathcal{P}$ | $\mathcal{F}$ | $\pi(A,F)$ | similarity or dissimilarity computation $d(A,B)$ | Definitions of symbols used. See Section 4 for terminology | Clustering method |
|---|---|---|---|---|---|---|
| Hutchens and Basili, 85 | procedures | global variables | $\pi_r(A,F) =$ $A$ refers to $F$ <br> $\pi_a(A,F) =$ $A$ assignes to $F$ | recomputed dissimilarity: $\frac{s(A)+s(B)-2b(A,B)}{s(A)+s(B)-b(A,B)}$ | $e(A,F,B)=\pi_r(A,F)\wedge\pi_a(B,F)$ <br> $b(A,B)=\lvert e(A,-,B)\rvert+\lvert e(B,-,A)\rvert$ <br> $s(A)=\lvert e(A,-,-)\rvert+\lvert e(-,-,A)\rvert$ | Adaptation of HAC. Recomputes $b$, and $s$ at each step. |
| | | | | expected dissimilarity: $\frac{s(A)+s(B)}{(\dim(b)-1)\times b(A,B)}$ | dim $b$ is the row dimension of $b$ | |
| Selby and Basili, 91 | " " | " " | " " | similarity measure $b(A,B)$ | As defined above | single-link |
| Schwanke, 91 | procedures | global symbols procedures | $\pi_u(A,F) =$ $A$ uses $F$ <br> $\pi_c(A,F) =$ $A$ calls $F$ $\vee F$calls $A$ | similarity measure $\frac{w(r_A \cap r_B)+k\times\pi_c(A,B)}{n+w(r_A\cap r_B)+d\times(w(r_A-r_B)+w(r_B-r_A))}$ | $r_A=\{F:\pi_u(A,F)\}$ <br> $w(X)=\sum_{x\in X} -log(Pr(x))$ <br> $Pr(x)=\lvert\pi_u(-,x)\rvert/\lvert\pi_u(-,-)\rvert$ <br> $n$, $k$, and $d$ are user defined parameters | Adaptation of HAC. Interactive |
| Maarek et. al., 91 | programs | pairs of words in documentation | $\Pi_w(A,F) =$ Number of times $F$ appears in $A$ | dissimilarity measure $\sum_{i\in a_A\cap a_B} \hat{\rho}_A(i)\times\hat{\rho}_B(i)$ | $a_A=\{F:\Pi_w(A,F)>0\}$ <br> $\rho_A(F)=-\Pi_w(A,F)\times\log(Pr(F,A))$ <br> $Pr(F,A)=\Pi_w(A,F)/\sum\Pi_w(A,-)$ <br> $\hat{\rho}_A=(\rho_A-\bar{\rho_A})/\sigma_{\rho_A}$ <br> $\bar{\rho_A}$ : mean of $\rho_A$ <br> $\sigma_{\rho_A}$ : standard deviation of $\rho_A$ | single-link or complete-link |

**5.1.2 ART of Selby & Basili [SB91]** The ART of Selby & Basili use the same RUD as the ARTs of Hutchens & Basili, described above. It uses the single-link HAC with a similarity matrix consisting of the binding matrix, $b$, as defined by Hutchens & Basili [HB85].

$$sim_3(A, B) = b(A, B)$$

**5.1.3 ART of Schwanke [Sch91]** The RUD used by the ART of Schwanke [Sch91] consists of the $\pi_c$ relation (which procedure calls which procedure) and the $\pi_u$ relation (which procedure uses which global variable). The similarity matrix it creates may be stated as:

$$sim_4(A, B) = \frac{w(r_A \cap r_B) + k \times \pi_c(A, B)}{n + w(r_A \cap r_B) + d \times (w(r_A - r_B) + w(r_B - r_A))}$$

where:

- $n$, $k$, and $d$ are user defined parameters,
- $r_A = \{F : \pi_u(A, F)\}$, i.e. the set of global variables used by procedure $A$,
- $w(X) = \sum_{x \in X} - log(Pr(x))$, i.e. the weight associated to (or the discriminating power of) a set of global variables, and
- $Pr(x) = |\pi_u(-, x)|/|\pi_u(-, -)|$, the proportion of use relations involving global variable $x$ (or the probability that a use relation involves variable $x$).

Schwanke's ART uses the single-link HAC algorithm for creating clusters. It also provides two interactive interfaces to validate the clusters being created. In the first interface, after every cluster is created the user is asked to confirm it. In the second interface, a new cluster is first validated by using a heuristic; if the validation fails the user is queried. The heuristic used by Schwanke is: if the program symbols being grouped are declared in the same file then the grouping is okay, else it is not. Implementation of this heuristic requires the $\pi_d$ relation, that is which program element is declared in which file. This extends the information needed to create the RUD used by Schwanke's method.

**5.1.4 ART of Maarek et. al. [MBK91]** Unlike all the other ARTs studied in this paper, Maarek et. al.'s ART does not aim at classifying a set of procedures, variables, or types into modules. Instead, it aims at grouping related "programs" into groups of software libraries. Though not explicitly stated by the authors, this technique could well be used for architecture recovery. This technique differs significantly from other numeric, stratified techniques in that it uses information from program documentation, not the code itself. It uses a multigraph RUD, representing the $\Pi_w$ relation between a program's documentation and its lexical affinities. Maarek et. al. suggest using either a single-link or complete-link HAC with the dissimilarity matrix created by the following function:

$$diff_5(A, B) = \sum_{i \in a_A \cap a_B} \hat{\rho}_A(i) \times \hat{\rho}_B(i)$$

where:

- $a_A = \{F : \Pi_w(A, F)\}$, i.e. the set of lexical affinity found in document $A$,
- $\rho_A(F) = -\Pi_w(A, F) \times \log(Pr(F, A))$, i.e. resolving power of lexical affinity $F$ in document $A$,
- $Pr(F, A) = \Pi_w(A, F)/\sum \Pi_w(A, -)$, probability that lexical affinity $A$ appears in the documentation of program $F$, and
- $\hat{\rho}_A = (\rho_A - \bar{\rho_A})/\sigma_{\rho_A}$, normalized resolving power. $\bar{\rho_A}$ is the mean of $\rho_A$ and $\sigma_{\rho_A}$ is its standard deviation.

## 5.2 Numeric, non-stratified ARTs

Numeric, non-stratified ARTs are techniques that use numerical computation to group program symbols into modules. They are non-stratified in that they only create one set of groups, not levels of groups. The techniques of [BE81, PCB92] fall in this category. The first one creates partitioned modules while the second one creates overlapping modules. Like other numeric ARTs, these methods compute a matrix of values using information contained in an RUD. They differ in the relations they use to create an RUD, the computations used to create the respective matrices, and the functions used to identify groups.

#### 5.2.1 ART of Belady & Evangelisti [BE81]

Belady & Evangelisti use the $\pi_u$ relation, i.e. which function uses which global variable*. The similarity matrix they use corresponds to the adjacency matrix representation of $\pi_u$.

$$sim_6(A,B) = \begin{cases} \pi_u(A,B) \rightarrow & 1 \\ \neg\pi_u(A,B) \rightarrow & 0 \end{cases}$$

In creating this matrix, they require that the procedures be assigned row (and column) numbers lower than the global variables. This matrix is then input to a clustering algorithm due to Donath [DH72]. It takes two parameters: the number of clusters, $N$, to be generated and the maximum number of nodes allowed in each cluster. A cluster, hence a module, may have both procedures and global variables. Donath's algorithm generates $N$ eigenvectors using the similarity matrix. These eigenvectors are then used to place each node in one of the $N$-modules. The details of the placement algorithm may be found in the original paper.

#### 5.2.2 ART of Patel et. al. [PCB92]

Patel et. al.'s ART uses $\Pi_t$, the multigraph RUD representing how many times a procedure refers to a type. It creates a similarity matrix using the following function:

$$sim_7(A,B) = \frac{\vec{\Pi_t}(A) \times \vec{\Pi_t}(B)}{\left\|\vec{\Pi_t}(A)\right\| \times \left\|\vec{\Pi_t}(B)\right\|}$$

where $\vec{\Pi_t}(A)$ is a vector representing $\Pi_t(A,-)$ and all vectors use the same permutation to for assigning position to the counts for the types. The vector product, therefore, represents the computation:

$$\vec{\Pi_t}(X) \times \vec{\Pi_t}(Y) = \sum_{a \epsilon T} \Pi_t(X,a) \times \Pi_t(Y,a)$$

and the vector dimension represents the computation:

$$\left\|\vec{\Pi_t}(X)\right\| = \sqrt{\sum_{a \epsilon T} \Pi_t(X,a)^2}$$

where $T$ is the set of all types used in the program.

Patel et. al.'s ART is not constructive, in that it does not generate a set of groups representing modules. Instead, it provides a function to test if a set of procedures constitute a module. A set of procedures $S$ constitute a module if $T(S) \geq \tau$, where $\tau$ is some experimentally determined threshold value and $T(S)$ is defined as:

$$T(S) = \frac{\sum_{x,y \in S; x \neq y} sim_7(x,y)}{|S| \times |S-1|}$$

### 5.3 Graph-theoretic, stratified ARTs

Only the ART of Choi & Scacchi [CS90] falls into this category. Unlike the other ARTs discussed so far, this uses an RFD $\psi$ (not an RUD) to recover an architecture. Further, it uses graph-theoretic computation and organizes a program's symbols in a hierarchy. This hierarchy is similar to that due to HAC based ARTs, in that the program symbols appear only at the leaf and that each symbol appears at most once. The intermediate nodes are newly added abstract nodes representing modules. Choi & Scacchi's ART, therefore, creates partitioned modules. The hierarchy it generates is different from dendrograms in that there is no level number associated to the nodes.

Choi & Scacchi's ART depends on finding the biconnected components of an undirected graph. A *biconnected component* of a graph is its subgraph whose every pair of distinct edges lie on some cycle. A node may be in more

* Belady & Evangelisti [BE81] actually use the relation "which function uses which control block." In their context, a control block corresponds to a global variable, hence we say that they use the $\pi_u$ relation.

than one biconnected component, but any two biconnected components may have at most one node in common. Nodes common to different biconnected components are called *articulation points*. Algorithms for finding the biconnected components of a graph may be found in texts on algorithm analysis, such as [AHU74].

Choi & Scacchi's ART finds the biconnected components of an RFD. It then creates a module for each articulation point. Each module consists of an articulation point and submodules created by applying the algorithm recursively to the subgraphs induced by the vertices of each biconnected component, except the articulation points. Choi and Scacchi argue that this procedure extracts an architecture with minimum alteration distance (which is the same as the sum of all cladistic distances) and minimum coupling (sum of the number of children of all nodes).

## 5.4 Mixed, stratified ARTs

Müller & Uhl's [MU90] is the only ART we have studied that uses a combination of graph-theoretic and numeric computations and generates stratifications that are not hierarchies. It uses a multigraph RFD $\Psi$ to compute the similarity matrix. Its algorithmic structure can also be abstracted as an HAC (i.e., iteratively select similar elements, combine them, and create a new similarity matrix). It provides four methods to decide if pairs of elements belong to the same group. The technique is interactive, in that the onus of choosing the appropriate method in each iteration lies with the user. Unlike HACs it also provides methods to decide if two elements do not belong to the same group. The four methods of choosing program elements that may be grouped are:

1. selection by interconnection strength
2. selection by centricity
3. selection by common neighbors, and
4. selection by name

The first three selection methods use numeric computations similar to those used by HAC based methods. These computations are, however, best expressed using graph-theoretic concepts (hence its classification as mixed).

The interconnection strength measure between two nodes $A$ and $B$ is the exact number of syntactic objects exchanged between the two nodes, i.e.

$$sim_8(A, B) = \Psi(A, B)$$

Müller & Uhl classify two components to be strongly related if their interconnection strength is greater than a certain threshold $T_h$ and loosely related if it is less than a certain threshold $T_l$. Components with strong interconnection strength are placed in the same group and those with different interconnection strength are placed in different groups.

Müller & Uhl and suggest two similarity measures for selecting on common neighbors, one based on common successors common successors between the pair:

$$sim_9(A, B) = |\pi_\psi(A, -) \cap \pi_\psi(B, -)|$$

and the other on their common predecessors:

$$sim_{10}(A, B) = |\pi_\psi(-, A) \cap \pi_\psi(-, B)|$$

Remember that $\pi_\psi(a, b) = |\Psi(a, -, b)| > 0$. Two elements $A$ and $B$ are placed in the same module either if $sim_9(A, B) \geq T_c$ or $sim_{10}(A, B) \geq T_s$, where $T_c$ and $T_s$ are two threshold values.

To select by centricity, the exact interface or the number of dependences between an element and other elements is computed.

$$ei(A) = \Psi(A, -) + \Psi(-, A)$$

This is then used to identify central and fringe nodes defined as those with the exact interface beyond the thresholds $T_k$ and $T_f$, respectively. Such elements are assigned to different groups.

Selection by name is unique to Müller & Uhl's ART. Two program elements are considered similar if their names have matching substrings (e.g. common prefix).

After grouping two elements, Müller & Uhl's ART creates a new dependency matrix by replacing the pair of similar elements with a new node. Its dependencies with the other elements is computed by adding the sum of the dependencies of the elements grouped. Müller & Uhl allow multiple selection operations to be applied to the same dependence matrix. This means that an element or a module may be included in more than one module leading to a non-hierarchical, yet stratified, architecture. However, if during any iteration only one selection operation is applied then the architecture would be hierarchical.

Though, Müller & Uhl's ART can be abstracted as an HAC, the architectures are not dendrograms as there are no level numbers associated to each cluster. Since the technique uses several different similarity computations there is no trivial way to use similarity values as levels.

Another important question when using this technique is: How does one define the various thresholds? This question becomes moot since Müller & Uhl's ART allow the threshold values to be set interactively and changed between iterations.

## 5.5 Graph-theoretic, non-stratified ART

Liu & Wilde [LW90] present two ARTs to group a set of functions, types, and global variables into modules. These ARTs use graph-theoretic computation and generate non-stratified architectures with overlapping modules. We refer to the two ARTs as *global based* and *type based.* The first technique uses an RUD consisting of the $\pi_u$ relation (i.e., which procedure uses which global variable) while the RUD used by the second technique consists of $\pi_i$ relation (i.e., which procedure uses which type in its interface).

The global based ART has two steps. In the first step the $\pi_u$ relation is used to create an undirected graph $(V, E)$ whose nodes represent global variables. In this graph there is an edge between two nodes if their exists a procedure that uses their corresponding global variables: i.e.,

$(x, y) \in E$ if $\exists p \cdot \pi_u(p, x) \wedge \pi_u(p, y)$.

Each strongly connected component of the graph $(V, E)$ represents a module. Let $C \subseteq V$ be the vertices in a strongly connected component. The set of program elements $M_1(C)$ in the corresponding module is defined by the following rules.

1. All the global variables represented by vertices of a strongly connected component are in its corresponding module, i.e. $C \subseteq M_1(C)$.
2. All the procedures using any global variable represented by the vertices of $C$ are in $M_1(C)$, i.e. $\forall g \in C\{p \mid \pi_u(p, g)\} \subseteq M_1(C)$.
3. No other symbols are in $M_1(C)$.

Since a procedure may be used by variables in multiple strongly connected components, it may be in multiple modules.

**Definition:** Let $t_1$ and $t_2$ be two types, $t_1 \ll t_2$ if type $t_1$ is a sub-type of $t_2$, i.e. $t_1$ is used to define $t_2$.

The type based ART first removes certain relations from $\pi_t$. If $\pi_t(p, t_1)$ and $\pi_t(p, t_2)$ are two relations, saying procedure $p$ uses types $t_1$ and $t_2$, then if $t_1 \ll t_2$, the relation $\pi_t(p, t_1)$ is removed. We call the new relation so created $\pi'_t$.

Modules are now identified by associating with each type $t$, $M_2(t)$ – a set of procedures and types, using the following rules.

1. All the procedures using type $t$ belong to this set, i.e. $\{p \mid \pi'_t(p, t)\} \subseteq M_2(t)$.
2. All the types used by any of these procedures are also in this set, i.e. $\{t_2 \mid \pi'_t(p, t) \wedge \pi'_t(p, t_2)\} \subseteq M_2(t)$.
3. No other symbols belong to $M_2(t)$.

Once again, a procedure may be in more than one module since it may use more than one type.

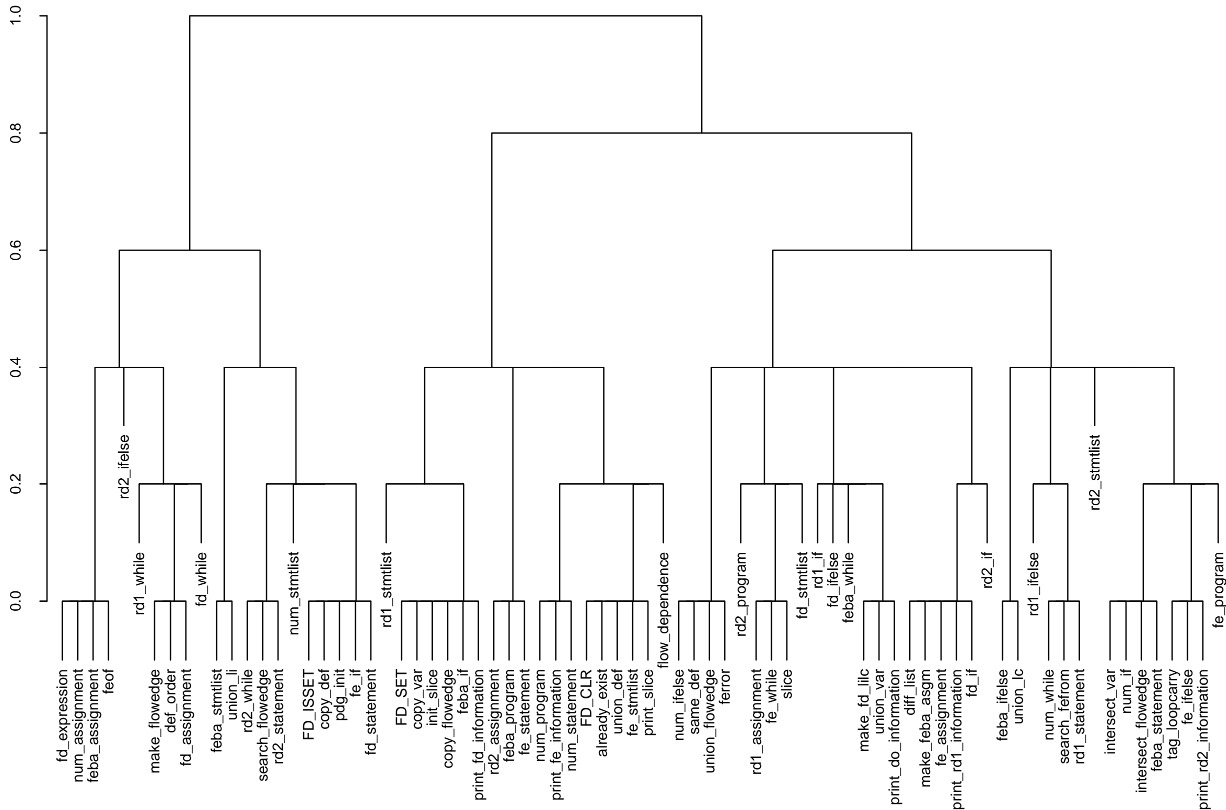



Figure 4 An architecture of parts of a program slicing system recovered using single-link HAC algorithm. The strings “rd1”, “rd2”, “fe”, “feba”, and “fd” are acronyms for different type of data and control flow analyses. The strings “if”, “while”, “ifelse”, “assignment”, etc. denote different program construct. The system used the convention of naming functions by combining these strings. Notice the above architecture does not do a good job of identifying related functions.

# 6 Comparing architectures

## 6.1 Why compare architectures?

While the ARTs surveyed in Section 5 represent significant research progress, in the absence of any objective measure to evaluate the quality of architectures recovered, their validation has been subjective. This is exemplified by the following extracts:

> “Although we performed the analysis with no foreknowledge of the system, our derived structure is consistent with the mental image held by the chief maintainer of the Practice Manager.” [MMM90, Section 8]

and

> “There was a close correspondence between the two views of the system. This may be seen by the dendrogram of the smaller system in the Appendix. The two capital letters preceding each of the Fortran procedure names designate the subsystem in which the designers placed the routine.” [HB85, Section III.E.2, page 755]

The quotations suggest that the respective architecture recovery techniques were validated on software systems for which the expected architecture could be defined by either its maintenance programmer (first case) or by some naming convention (second case). The comparison of the recovered architecture with the expected architecture was, however, done manually by having the designers or maintainers visually inspect the graphs created by each method and assess if they matched their view or mental model of the systems' module hierarchy. Similar validation procedures have been used by other works too.

The graph representation of a stratified architecture for a large system can be quite complex. For instance, the dendrogram referred to in the second quotation has 246 leaf nodes and the number of intermediate nodes are equally large. Checking for its "correctness" – how well it matches an expected module decomposition – by inspection is prone to error. The complexity of this exercise can be experienced by looking at the dendrogram of Figure 4. Even a small variation in a graph, that may easily go undetected by a human, could change its "meaning" drastically. There is, therefore, a need to automate the comparison of the recovered architecture with an expected architecture. The latter may be derived from clues provided by the names of the symbols or other system specific information.

## 6.2 Properties of a congruence measure

The measures for cluster difference given in Section 3.3, the $\Delta$ functions, can not be used directly for comparing architectures. The problems are as follows:

1. The $\Delta$ functions only compare dendrograms. But, the expected module hierarchy used for measuring the degree of correctness and the architectures recovered by Choi and Scacchi's technique [CS90] are not dendrograms.
2. The $\Delta$ functions compare dendrograms over the same set of elements. Since the architecture recovery techniques use different structural information, the architectures recovered by them may not contain the same program elements.

Furthermore, the $\Delta$ functions measure the difference between two clusters. They return a high value when two clusters are very different and a low value when the clusters are similar. Therefore, when measuring correctness of a technique, say by taking the mean of several $\Delta$ values, a high value indicates low degree of correctness.

We prefer a measure of congruence, $\mu : DG \times DG \rightarrow \mathcal{R}_{01}$, between clusters that returns a high value when two architectures are close to each other and low when they are apart; the inverse of 'difference'. The function $\mu$ must satisfy the following properties.

1. Its range should from $0$ to $1$.
2. If it returns a $1$ it should indicate that the two architectures are completely congruent, i.e.:
   a. the two architectures organize the same set of elements and
   b. the dissimilarity coefficient mapping of their dendrograms is identical.
3. If it returns a $0$ it should indicate that the architectures have nothing in common, i.e.
   a. the two architectures have at most one element in common or
   b. the two architectures are flat partitions and all pairs of elements in the two architecture are 'differently placed'.

   Two elements are said to be *differently placed* if they are in the same partition in one architecture but in different partitions in another [Ran71]. The two extreme values capture the extreme possibilities when comparing architectures.
4. If two architectures organize the same set of elements then it should return a higher number when they have fewer differently placed elements.
5. If none of the elements organized by both the architectures are differently placed then it should return a higher number when the fraction of elements common to both is higher.

## 6.3 Measure of congruence of architectures

We solve the first problem by choosing two different $U$ functions (mapping of a cluster to dissimilarity coefficients) measuring analogous properties, one for dendrograms and one for trees. The dissimilarity coefficients using the appropriate $U$ function may then be used in the $\Delta$ functions. The second problem is solved by comparing only the architectures defined over elements in the intersection of two architectures and multiplying the resulting 'difference' by an *intersection ratio*. The details are given below.

As stated earlier, a tree can be interpreted as a dendrogram by equating the depth of each node to the level of a dendrogram. Choi and Scacchi's algorithm minimizes the sum of the cladistic distance between all pairs of leaves in the recovered architecture. It is appropriate, therefore, to use the function $U_3$ (introduced in Section 3.3) to map their tree into a dissimilarity matrix. But this function is not appropriate for traditional dendrograms because it assumes a unit distance between two levels irrespective of their difference. There is no obvious analogy between $U_1$ and $U_3$, hence it would not be appropriate to compare values resulting from these functions.

An alternative is to define, for dendrograms, a function $U_4$, that measures cladistic distance between pairs of elements using level values. This requires introducing a function $L : DG \rightarrow P \rightarrow R_+$, the *lowest level partition* in the dendrogram $c$ at which an element is first clustered with a different element.

**Definition:** $L(c) = (\lambda A) \cdot (min\{h : \exists B, B \neq A, (A, B) \in c(h)\})$

Notice that the expression $U_1(c)(A, B) - L(c)(A)$ is the difference between the lowest level at which $A$ is clustered with some other element and the lowest level at which both $A$ and $B$ are in the same cluster. This is analogous to cladistic distance – the number of edges traversed to reach the leaf node $A$ from the leaf node $B$. The following function $U_4$ is then analogous to $U_3$.

**Definition:** The $U_4 : DG \rightarrow P \times P \rightarrow R_+$, between pairs of elements in a dendrogram is then defined as: $U_4(c) = (\lambda A, B) \cdot ([U_1(c)(A, B) - L(c)(A) + U_1(c)(A, B) - L(c)(B)]/2)$

The sum of level differences is divided by two to normalize it.

**Definition:** Let $E : DG \rightarrow 2^{\mathcal{P}}$ be the set of all the elements in the recovered architecture $c$, i.e. $E(c) = \{A : \exists h, B, (A, B) \in c(h)\}$.

That two architectures $c_1$ and $c_2$, of the same program recovered using two different techniques do not organize the same set of elements may be stated as $E(c_1) \neq E(c_2)$. Using the $\Delta$ functions one can only compare the architectures over the set $E(c_1) \cap E(c_2)$ as defined by the projection of one architecture over the other.

**Definition:** The *projection*, $_ \mid _ : DG \times DG \rightarrow DG$, of architecture $c_1$ with respect to architecture $c_2$ is defined as: $(c_1 \mid c_2)(h) = c_1(h) \cap E(c_1) \times E(c_2)$.

In other words, $c_1 \mid c_2$ is the same as $c_1$ at each level except that relations involving elements not in $c_2$ are removed. The congruence of $c_1 \mid c_2$ and $c_2 \mid c_1$ may be measured using one of the $\Delta$ functions stated earlier. But such a measure will be insensitive to the elements in the two architectures that are not in the other. The inverse of this is measured by the *intersection ratio* given below.

**Definition:** The intersection ratio, $I : DG \times DG \rightarrow R_+$, of architectures $c_1$ and $c_2$ is:

$$I(c_1, c_2) = |E(c_1) \cap E(c_2)| / |E(c_1) \cup E(c_2)|$$

The intersection ratio is $1$ when both the recovered architectures organize the same set of elements. It is $0$ if they organize entirely different set of elements. Therefore, a high intersection ratio is good and a low intersection ratio is bad. The congruence of two architectures may now be measured as follows.

**Definition:** The *measure of congruence*, $\mu : DG \times DG \rightarrow \mathcal{R}_{01}$, of architectures $c_1$ and $c_2$ is given by: $\mu(c_1, c_2) = I(c_1, c_2) \times (1 - \Delta(U_j(c_1 \mid c_2), U_j(c_2 \mid c_1))$ where $j \in \{2, 4\}$.

In our initial experiments [LMP92], we derived a function equivalent to $\Delta_3$ before being exposed to Jardine and Sibson's [JS71] work. This is, therefore, our choice of the difference measure when computing congruence. Generally speaking one may choose any of the $\Delta$ functions given in Section 3.3 or devise a new one. For a meaningful comparison of architectures due to Choi & Scacchi's ART with those of other ARTs the $U_j$ in computing congruence should, however, be restricted to $U_2$ for trees and $U_4$ for dendrograms.

# 7 An experiment with Hutchens-Basili ARTs

During Summer'92 we conducted an experiment to measure the correctness of Hutchens & Basili's ARTs [LMP92, HB85]. The intent of our experiment was to measure how successful these ARTs were in recovering the actual architecture of a set of systems that were designed using data abstraction [Par76]. The most important parameter for such an experiment is the programs analyzed. Ideally, the programs should satisfy the following constraints:

1. The programs should have diligently used data abstraction and the module decomposition should be easily retrievable from the source code.
2. The programs should be representatives of *real-world* programs.

These two constraints are hard to satisfy simultaneously; since it is hard to find a set of programs that are representatives of real-world programs and use data abstraction. The retrievability of the design decomposition from the code is important to use as an "oracle" for the expected architecture.

Given the difficulty in choosing subject programs, it was our choice to give higher precedence to the first constraint. We, therefore, used programs developed in our course "Introduction to Software Engineering." These programs were ensured to satisfy the first constraint since their development was rigorously controlled. All the programs had the same abstract data type based design and hence the same module decomposition[†]. This was ensured since a) the design was provided to the students (as a routine part of the instruction) and b) the design violations were removed over four iterations, one per week. During the iterations, that the programs implemented the given design was verified by a) review of each program by the grader, b) review of each module by two fellow students, and c) generation of "program families" (*a la* Parnas) by exchanging modules between students.

The last method of verification provided the most rigorous check since each student generated nine programs by mixing modules from two other students. If a module violated the interface constraint, there was a high likelihood that it would cause compile-time or run-time failures when combined with someone else's module.

The author along with two graduate students implemented the two ARTs proposed by Hutchens & Basili [HB85] (see Table 2) with three variations of $\pi$, and measured the congruence of the recovered architectures of 19 C programs varying in size from 500 lines (including comments) to 2400 lines.

In computing the measure of congruence $\mu$ we used the $\Delta_3$ function to measure the difference between architectures. It was our observation that a) most of the recovered architectures had several misplaced procedures or did not organize all the procedures and b) the congruence measures were higher when the recovered architectures found more correct modules and lower when fewer correct modules were found. The congruence measures, therefore, correctly reflected how close a recovered architecture was to the expected one. Figure 5 plots the congruence measures for variations of the two ARTs proposed by Hutchens & Basili. The variations introduced, in both cases, was that the relation $\pi_u$ was used to substitute $\pi_r$ and $\pi_d$. Accordingly, the set $\mathcal{F}$ was redefined to be the set of all global symbols. The means of the congruence measures of the plots are $0.52$ and $0.53$ for expected and recomputed methods, respectively. On a range of 0 to 1, this is not a very high value. However, the congruence measures for unmodified Hutchens & Basili's ARTs were worse, with means of $0.21$ and $0.22$, respectively.

# 8 Conclusions and future work

Software is an "immortal asset" of an organization [Bus89]. The capital budget required to redevelop many software systems is simply too large to justify scrapping them [You92]. To keep up with changes in technology, re-engineering these systems is the only viable alternative. Any re-engineering task involves reverse engineering, i.e. recovering of design and other abstractions from source code. Arnold's compilation of papers on Software Re-engineering [Arn93] and the proceedings of the *IEEE Conference on Software Engineering* is rife with re-engineering exercises in companies such as Anderson Consulting, DEC, IBM, Lockheed, and NASA. In these

[†] Though it is common to use programs with identical specifications in software engineering experiments [BGS84, HB85], to the best of our knowledge, this is the first instance of an experiment with programs implementing identical design.

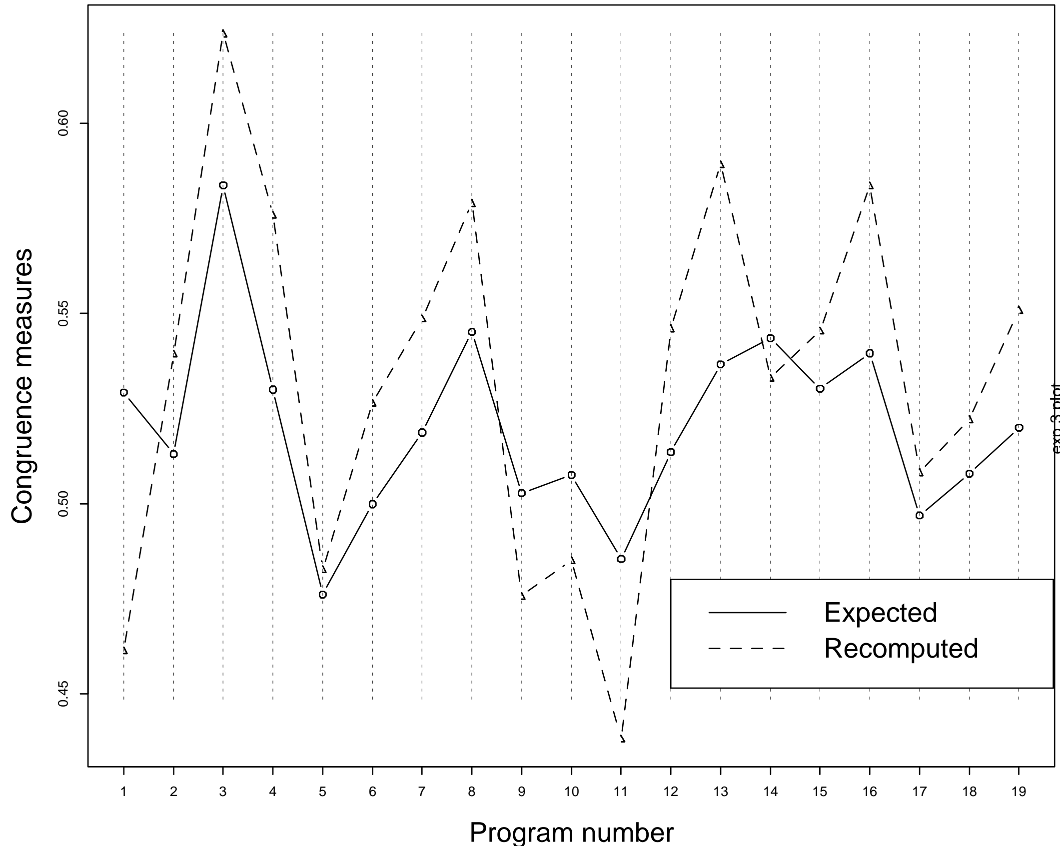



Figure 5 Overlayed plot of congruence measures for architectures recovered for 19 assignment programs using variations of Hutchens and Basili's architecture recovery techniques [HB85].

exercises researchers have developed various techniques to recover software architectures, i.e. its decomposition in modules.

So far there has been no easy way to evaluate a technique's ability in recovering a correct architecture or comparing the performance of two techniques. The bottleneck has been (a) the techniques are not presented using a consistent set of symbols and terms and (b) the lack of a measure to compare two architectures – an expected with a recovered or those recovered by two different ARTs. This paper fills both these needs. It presents a unified view of each technique by using a consistent set of symbols and terms and it presents a measure of congruence between architectures.

The work is significant since it now enables a comparison of ARTs based on the type of information they use, the kind of computation they perform, and the program elements that get included in an architecture they recover. A comparative knowledge of such information may help in a) deciding whether a particular technique would be applicable in a given context, b) whether a cross reference information extraction tool [BHW89, CNR90, Rei90, WCW88] can be used to build an ART, and c) creating new ARTs by combining elements from different ARTs.

To cite a few examples. ARTs that use 'type' related information would not be useful with old FORTRAN programs since these programs do not have user defined types. ARTs that use 'global variables' related information may not do well with programs using data abstractions. To develop Liu & Wilde's type-based ART [LW90] and Patel et. al.'s ART [PCB92] one needs the information 'which type is used to define which type.' A cross-reference information extraction tool that does not provide this information would not be suitable for this purpose. Patel et. al.'s ART also requires the count of how many times a relation between two symbols exists and hence the tool used should permit such a computation. When a technique is not directly applicable in a particular context, it may

be adapted using a different set of relations. For instance, we modified Hutchens & Basili's ARTs [HB85] to use relations of procedures with types instead of with global variables. The modified ARTs performed better than the originals, as determined by the congruence measure, for programs using data abstraction.

An ambitious experiment that we plan to carry out, subject to availability of resources, is to identify the factors that influence the correctness of architecture recovered by various techniques. For instance, we hypothesize that the following program properties may have influence on the architecture recovered by an ART:

F1. choice of data structures for implementing an abstract data type (such as a sequence may be implemented by a list or an array),
F2. implementation decisions (such as defining a special *typedef* for data types processed by a module or using the name of a primitive type used to implement it), and
F3. size of the program (in lines of code or any other measure).

Since factors F1 and F2 influence the cross-reference relations used by some of the ARTs , it should influence the architectures they recover. For example, to implement a sequence as a list requires defining a *struct* (in C terminology) which introduces additional symbols for its fields; these symbols are not introduced in an array based implementation. Similarly, F2 influences the number of global symbols in a system and hence may influence the recovered architecture. The recovered architecture may potentially depend on the size of the relations which may depend on the size of the program, hence F3.

We fathom that successful identification of factors influencing the recovered architecture can lead to the development of (a) methods for estimating the quality of the recovered architectures and (b) guidelines to "calibrate" recovery techniques using program related properties. The estimation method may be developed by measuring the dependence on factors that have been identified as influencing the recovered architectures. The calibration of a technique, such as that of Schwanke's [Sch91], would involve varying parameters of the clustering method based on properties of programs under study so as to recover architectures closest to the module hierarchy. The guidelines and estimation methods could be in the form of standardized tables like those used by engineers.

#### Acknowledgments

This work made use of the FIELD programming environment under license from Brown University. We acknowledge Steve Reiss its development. The work was partially supported by the grant LEQSF (1993-95) RD-A-38 and conducted in Software Research Laboratory established by the grant LEQSF (1991-92) ENH-98, both from the Louisiana Board of Regents.

**Note to referees:** An ART has also been proposed by Ong & Tsai [OT93]. It has not been studied in this paper, but will be in the final manuscript. The work came to my attention just a day before this manuscript was sent to the editors.

# Bibliography

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachussets, 1974.

[Ale64] Christopher Alexander. *Notes on the synthesis of form*. Harvard University Press, Cambridge, Massachusetts, 1964.

[Arn93] Robert S. Arnold. *Software Reengineering*. IEE Computer Society Press, Los Alamitos, California, 1993.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BE81] L. A. Belady and C. J. Evangelisti. System partitioning and its measures. *Journal of System and Software*, 2(1):23–29, February 1981.

[BGS84] Barry W. Boehm, Terence E. Gray, and Thomas Seewaldt. Prototyping versus specifying: a multiproject experiment. *IEEE Transactions on Software Engineering*, SE–10(3):290–302, May 1984.

[BHW89] T. J. Biggerstaff, Josiah Hoskins, and Dallas Webster. DESIRE: A system for design recovery. Technical Report STP-081-89, MCC/Software Technology Program, April 1989.

[Bus89] Eric Bush. A CASE for existing systems. Technical report, Language Technology, Salem, MA, 1989.

[Chi90] Elliot J. Chikofsky. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, January 1990.

[CNR90] Yih-Farn Chen, Michael Y. Nishimoto, and C. V. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.

[CS90] Song C. Choi and Walt Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, pages 66–71, January 1990.

[Day79] William H. E. Day. The complexity of computing metric distances between paritions. Technical Report 7901, Department of Computer Science, Memorial University of Newfoundland, Newfoundland, Canada, September 1979.

[DH72] W. E. Donath and A. J. Hoffman. Algorithms for partitioning of graphs and computer logic based on eigenvectors of connection matrices. *IBM Tech. Disclosure Bull.*, 15(3), 1972.

[Eve74] B. Everitt. *Cluster Analysis*. Heinemann, London, England, 1974.

[Far79] J. S. Farris. A successive approximation approach to character weighting. *Syst. Zool.*, 18:374–385, 1979.

[HB85] David H. Hutchens and Victor R. Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, pages 749–757, August 1985.

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs*. North-Holland, New York, 1977.

[JL91] Ivar Jacobson and Fredrik Lindstrom. Re-engineering of old systems to an object-oriented architecture. In *Proc. OOPSLA*, pages 340–350, 1991.

[JS71] N. Jardine and R. Sibson. *Mathematical Taxonomy*. John Wiley and Sons, Inc., New York, 1971.

[LB85] M. M. Lehman and L. A. Belady. *Program Evolution*. Academic Press, 1985.

[LMP92] Arun Lakhotia, Sanjay Mohan, and Pruek Poolkasem. On evaluating the goodness of architecture recovery techniques. Technical Report CACS-TR-92-5-4, University of Southwestern Louisiana, Lafayette, September 1992.

[LS80] B. Lientz and E. Swanson. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley, Reading, MA, 1980.

[LW90] S. Liu and N. Wilde. Identifying objects in a conventional procedural language: An example of data design recovery. In *Proc. IEEE Conference on Software Maintenance*, pages 266–271, November 1990.

[MBK91] Y. S. Maarek, Daniel M. Berry, and Gail E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, August 1991.

[MJ81] Steven Muchnick and Neil Jones. *Program Flow Analysis: Theory and Applications*. Prentice-Hall,Inc., 1981.

[MMM90] Hausi A. Müller, Jochen R. Mohr, and James G. McDaniel. Applying software re-engineering techniques to health information systems. *Proceedings of the IMIA Working Conference on Software Engineering in Medical Informatics (SEMI), Amsterdam*, October 1990.

[MS83] Ryszard S. Michalski and Robert E. Stepp. Automated construction of classifications: Conceptual clustering versus numerical taxonomy. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(4):396–409, July 1983.

[MU90] Hausi A. Müller and James S. Uhl. Composing subsystem structures using (K,2)–partite graphs. *Proceedings of the Conference on Software Maintenance*, pages 12–19, November 1990.

[OT93] C.L. Ong and W. T. Tsai. Class and object extraction from imperative code. *J. Object Oriented Programming*, pages 58–68, Mar–Apr 1993.

[Par76] David L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1), March 1976.

[Par86] G. Parikh. *Handbook of Software Maintenance*. Wiley-Interscience, New York, N.Y., 1986.

[PCB92] Sukesh Patel, William Chu, and Rich Baxter. A measure for composite module cohesion. In *Proceedings of the 14th International Conference on Software Engineering*, pages 38–48, 1992.

[Ran71] William M. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, 66(336):846–850, December 1971.

[Rei90] Steven P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66, July 1990.

[Roh74] F. James Rohlf. Methods of comparing classifications. *Annual Rev. Ecol. Syst.*, 5:101–113, 1974.

[SB91] Richard W. Selby and Victor R. Basili. Analyzing error-prone system structure. *IEEE Transactions on Software Engineering*, pages 141–152, February 1991.

[Sch87] N. Schneidewind. The state of software maintenance. *IEEE Transactions on Software Engineering*, 13(SE–3):303–310, March 1987.

[Sch91] R. Schwanke. An intelligent tool for reengineering software modularity. In *Proc. 13th International Conference on Software Engineering*, 1991.

[SR62] Robert R. Sokal and F. James Rohlf. The comparison of dendrograms by objective methods. *TAXON*, XI(2):33–40, 1962.

[SS94] Ricky E. Sward and Robert A. Steigerwald. Issues in re-engineering from procedural to object-oriented code. In Bruce I. Blum, editor, *Proc. of the Fourth Systems Re-engineering Technology Workshop*, pages 327–333. John Hopkins University Applied Physics Laboratory, 1994.

[WCW88] A. Wolf, L. Clarke, and J. Wileden. A model for visibility control. *IEEE Transactions on Software Engineering*, pages 512–520, April 1988.

[You92] Edward Yourdon. *Decline and Fall of the American Programmer*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1992.