A unified framework for expressing software subsystem classification techniques

Arun Lakhotia

Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504
arun@cacs.usl.edu
(318) 482-6766, Fax: -5791

Accepted for publication in the J. of Systems and Software. To appear in 1996.

Abstract

The architecture of a software system classifies its components into subsystems and describes the relationships between the subsystems. The information contained in such an abstraction is of immense significance in various software maintenance activities. There is considerable interest in extracting the architecture of a software system from its source code, and hence in techniques that classify the components of a program into subsystems. Techniques for classifying subsystems presented in the literature differ in the type of components they place in a subsystem and the information they use to identify related components. However, these techniques have been presented using different terminology and symbols, making it harder to perform comparative analyses. This paper presents a unified framework for expressing techniques of classifying subsystems of a software system. The framework consists of a consistent set of terminology, notation, and symbols that may be used to describe the input, output, and processing performed by these techniques. Using this framework several subsystem classification techniques have been reformulated. This reformulation makes it easier to compare these techniques, a first step towards evaluating their relative effectiveness.

1 Introduction

A software system is typically comprised of several interconnected components, such as procedures, functions, global variables, types, files, documentation, etc. An architecture of a software system classifies its components into subsystems and describes interactions between these subsystems [16, 33]. It provides a high level abstraction of the organization of a system and can be used to address the system level properties, such as capacity, throughput, consistency, and component capability [38].

Unless the architecture of a system is documented, which is very rare, its maintainer has to infer its overall structural organization from its source code. However, the architecture of a software system is not usually apparent from its source code. As a result there is considerable interest in developing automated support for recovering the architecture of a software system from its source code.

The crucial problem in recovering the architecture of a software system is classifying its components into subsystems. The interactions between subsystems may be inferred from the interaction between their member components. The subsystems must, of course, contain components that are related. Different subsystem classifications may be created by grouping components based on different types of relationships. For instance, *syntactic-units* such as functions, procedures, variables, and types may be organized so that units belonging to the same "module"*

* The term "module" is used in this paper in the sense used by Parnas [30] in the context of abstract data typing and information hiding.

are placed in the same subsystem [1, 6, 21, 27, 28, 31, 36]. Alternatively, one may place syntactic-units related to the same fault in the same subsystem [17, 37]. A subsystem may also consist of *collection-units*—files and modules—where units affected by the same change request are grouped together [23]. One may also classify a collection of complete programs into subsystems representing software libraries [22]. Subsystems may also be comprised of both syntactic and collection units [11, 25].

Research in subsystem classification techniques (SCT), as may be evident from the above description, has been influenced by different application needs, and has often progressed in isolation of the knowledge of others. This is reflected by the fact that SCTs have been presented using very different terminology and symbols. As a result it is difficult to compare and contrast various techniques. SCTs intrinsically depend upon heuristics which can be improved by borrowing ideas from other SCTs. Such an exchange of ideas is currently limited due to the lack of a unified framework within which SCTs may be described. A unified framework is also necessary to develop a sound foundation for what may be considered as a collection of *ad hoc* tools. This paper attempts to provide just that. The framework we develop allows the comparison of the inputs, processing, and outputs of various SCTs. It makes it easy to create new SCTs by mixing the strategies used in different SCTs and also enables the creation of meta-SCTs, i.e., SCTs parameterized over the choice of strategies.

The rest of the paper is organized as follows. Section 2 places research in SCT in the context of other research efforts. Section 3 presents our framework for expressing SCTs using a consistent set of notation, terminology, and symbols. Several of the known SCTs are reformulated in Section 4 using our framework. As a corollary, our reformulated description of a SCT may be significantly different from its description in the original paper. We do not attempt to highlight the differences and leave it for the reader to find the correspondence (or transformation) between our descriptions and that in the original works. Section 5 summarizes the benefits of a unified framework using examples from the context of the SCTs reformulated. Section 6 concludes the paper with some observations about why these heuristic techniques may be successful in identifying modules in legacy code.

2 Related works

The phrase "software architecture" has been used by researchers and practitioners for quite sometime now, although without a formal definition. The emerging field of software architectures aims at formalizing the notion of architectures and developing languages to specify architectures. In addition to describing the structure and topology of a system, these architecture specification languages will address system level properties such as capacity, throughput, consistency, and component capability [38]. Subsystem classification techniques may be used to automatically extract the architectures of existing systems and to document them using architecture specification languages.

The problem of classifying the subsystems a software belongs to the broader class of problems of recovering the design of a software system. A comprehensive survey of other design recovery techniques is beyond the scope of this work, and is unwarranted since it may be found elsewhere [5, 10, 41]. Subsystem classification is essentially a graph partitioning problem. Hence, it has much in common with VLSI layout problems. Some of the SCTs [6, 23] have in fact been influenced by algorithms for VLSI layout [13, 14].

Research in subsystem classification is also of significance to research in reengineering procedural programs into object-oriented programs [18, 40]. Such a reengineering task implicitly requires identifying a set of classes, their instance variables, and their methods, a requirement analogous to recovering the modular subsystems. Currently the identification of objects is done by analyzing a natural language description of the system using object-oriented analysis methods designed for forward engineering [12]. Object identification by processing a natural language is inherently non-automatable. It may be automated or semi-automated using SCTs instead [28]. Going one step further then, SCTs may be used to identify objects during forward engineering if the design or requirements are stated in a notation suitable for machine analysis. It is therefore not a coincidence that a recently proposed technique for deriving modular designs from formal specification [8] bears a resemblance to some of the SCTs discussed in this paper.

Software errors occur when a maintenance programmer's model of the architecture of a software system differs from the actual architecture. A tool has recently been proposed to reduce these errors by validating a programmer's

model [26]. Using this tool a maintenance programmer (a) postulates a set of subsystems of a program, (b) postulates the components that belong to that subsystem, and (c) postulates relationships between the subsystems. The tool then extracts the actual relationships between the subsystems by reflecting the relationships between the components onto the postulated subsystems. The similarities and the differences between the postulated relationships and extracted relationships highlight the discrepancies in the programmer's model of the system. This approach may be extended further by using SCTs to extract subsystem classifications and comparing the congruence between the extracted and the postulated subsystem classifications using a congruence metric [20].

3 A unified framework for subsystem classification techniques

In this section we develop a unified framework—terminology, notation, and symbols—to describe the inputs, outputs, and processing performed by SCTs. This framework is used in the next section to describe several SCTs.

3.1 Interconnection relations: Inputs to SCT

In order to recover the subsystem classification of a program, a SCT takes as input some interconnection relations [32] between the components of program. A program may have a variety of components, potentially related by several different interconnection relations. These relations may be represented as directed graphs with nodes representing program components and edges denoting interconnections between these components. (An edge emanates from a *source* node and terminates on a *target* node.) Different interconnection graphs relate different program components. We introduce the following generic symbols to categorize program components used in *an* interconnection graph:

- $\mathcal{P}$: the set of all the source nodes,
- $\mathcal{Q}$: the set of all the target nodes,
- $\mathcal{R}$: the set of program components tagged on edges, and
- $\mathcal{S} = \mathcal{P} \cup \mathcal{Q} \cup \mathcal{R}$: the set of all the program components in a graph.

Notice that these sets are generic in that their membership is defined with respect to a given interconnection relation.

The components of a program may be of several different "types," such as function, variable, procedure, file, document, etc. Instead of creating different sets of components for each of these types, we assume that each unique software component is represented by a unique abstract entity. The "type" (and also the "name") of a component is available as an attribute of this entity. This makes our sets homogenous without any loss of generality.

Interconnection relations provide the foundation for software maintenance tools, in general [7, 9, 35]. While there exist several interconnection relations (or graphs), those used by the SCTs studied in this paper can be classified into three categories, as follows.

Definition: An interconnection graph whose edges are tagged with numeric values is called a *weighted cross-reference graph* (WXG). A numeric value in these graphs denote the strength of the interconnection between the nodes (or components) connected by the corresponding edge.

Definition: An interconnection graph whose edges are tagged with a program component is called a *component flow graph* (CFG). An edge in a CFG represents a producer/consumer type relationship between the source and target node.

Definition: An interconnection graph whose edges do not have any tags is called a *component cross-reference graph* (CXG).

The CXGs, WXGs, and CFGs used by the SCTs described in this paper are summarized in Tables 1, 2, and 3, respectively. The symbols π, η, and ψ, respectively, denote the three types of graphs. These symbols are subscripted to identify a specific graph. The tables also describe the relation represented by each interconnection graph and their corresponding sets $\mathcal{P}$, $\mathcal{Q}$, and $\mathcal{R}$.

Table 1 Summary of component cross-reference graphs used by SCTs studied in this paper. See Section 3.1 for description of italicized phrases.

CXG π	**Type of components** **source** $\mathcal{P}$	**target** $\mathcal{Q}$	**Relation expressed by** $\pi(s_1, s_2)$
π_{su}	procedure	syntactic unit	syntactic unit s_2 appears somewhere in the declaration of procedure s_1
π_{ga}	procedure	global variable	procedure s_1 assigns to (i.e., modifies the value of) global variable s_2
π_{gu}	" "	" "	procedure s_1 uses (without modifying) global variable s_2
π_{gr}	" "	" "	procedure s_1 refers to (i.e., assigns or uses) global variable s_2 directly (i.e., without aliasing)
π_{gi}	" "	" "	procedure s_1 refers to (i.e., assigns or uses) global variable s_2 either direcly or indirectly (due to aliasing with a formal parameter)
π_c	" "	procedure	procedure s_1 calls procedure s_2
π_{ti}	" "	type	procedure s_1 has a *direct* or *downward reference* to type s_2 in its interface (i.e., type of formal parameters and return type).
π_{tl}	" "	" "	procedure s_1 has a local variable of type s_2
π_{tg}	" "	" "	procedure s_1 refers to a global variable of type s_2
π_{tt}	type	" "	type s_1 is used in defining type s_2 i.e., type s_1 is a *sub-part* of type s_2 or equivalently type s_2 is *super-part* of type s_1
π_{use}	statement block	variable	block s_1 uses variable s_2
π_{set}	" "	" "	block s_1 modifies variable s_2
π_{def}	" "	" "	block s_1 modifies variable s_2 before using it.
π_{live}	" "	" "	variable s_2 is live † at the end of block s_1.
π_d	file	syntactic unit	file s_1 contains a declaration of syntactic-unit s_2
π_f	" "	file	file s_1 includes file s_2

†: A variable is *live* after a block if it is used, before being modified, in some other block [3].

3.1.1 Notation used

Let s_1, s_2, and x represent program components. We use the notation $\pi(s_1, s_2)$ as a shorthand for $(s_1, s_2) \in \varepsilon(\pi)$, where $\varepsilon(\pi)$ gives the set of edges in the CXG π. The notation $\eta(s_1, s_2)$ gives the weight of the edge from component s_1 to component s_2 in the WXG η, where $\eta(s_1, s_2) = 0$ implies that there is no such edge. Analogously, $\psi(s_1, x, s_2)$ gives a boolean value denoting the absence or presence of an edge with tag x from component s_1 to component s_2 in the CFG ψ, i.e., component x flows from component s_1 to component s_2.

The Boolean value *true* is mapped to 1 and the value *false* to 0. Conversely, all positive natural numbers map to *true* and 0 to *false*. This mapping between the boolean and the natural number domains permits the use of predicates in arithmetic expressions and vice-versa.

Table 2 Summary of weighted cross-reference graphs used by SCTs studied in this paper.

WXG η	**Type of components** source $\mathcal{P}$	target $\mathcal{Q}$	**Relation expressed by** $\eta(s_1, s_2)$
η_w	program doc.	lexical affinity‡	Number of times lexical affinity s_2 appears in documentation of s_1
η_t	function	type	Number of times procedure s_1 has a direct or upward reference to type s_1

‡: Pairs of words are *lexical affinity* if they appear within ± 5 word distance [22].

Table 3 Summary of CFGs used by SCTs studied in this paper.

CFG ψ	**Type of components** source $\mathcal{P}$	tag $\mathcal{R}$	target $\mathcal{Q}$	**Meaning of** $\psi(s_1, x, s_2)$
ψ_{gr}	procedure	global variable	procedure	procedure s_1 modifies variable x, procedure s_2 uses variable x, and the definition of x in s_1 *reaches* s_2
ψ_{gc}	" "	" "	" "	procedure s_1 modifies variable x, procedure s_2 uses variable x, and there is an executable path from procedure s_1 to procedure s_2 (though the definition from s_1 does not reach the use in s_2)
ψ_{gx}	" "	" "	" "	procedure s_1 modifies variable x and procedure s_2 uses variable x, i.e., $\pi_{ga}(s_1, x) \wedge \pi_{gu}(s_2, x)$. (there need not be any execution path from procedure s_1 to procedure s_2)
ψ_F	file	syntactic units	file	Syntactic-unit (such as a variable, function, type) x is declared in file s_1 and used in file s_2

A '_' is used as a shorthand for an existentially quantified symbol, with each '_' in an expression representing a different existentially quantified symbol. The expression $\psi(_, B, _)$ is shorthand for $\{\psi(X, B, Y) : \exists X, Y \cdot \psi(X, B, Y)\}$.

Notice the difference between the two usages of $\psi(X, B, Y)$ above. In the first usage it is an unevaluated term. But in the second usage it is evaluated as a predicate that tests whether component B flows from component X to component Y. The difference is significant since $\psi(_, B, _)$ constructs a set. If for two pairs of X and Y, $\psi(X, B, Y)$ evaluates to *true*, there would be two entries in the set if the term is not evaluated but only one entry (*true*) if the term is evaluated. In a graph-theoretic interpretation $\psi(_, B, _)$ is the subgraph induced by edges with tag B in graph ψ.

This '_' notation may similarly be extended to be used as argument to π and η and used with set operators, such as $|\ |$ (length), $\cup$ (union), $\cap$ (intersection), and $-$ (set difference). A _ expression may also be used with arithmetic operators such as Σ. In this case the elements of the set are evaluated and the resulting values added. Thus, $\Sigma\eta(A,_)$ gives the sum of the weights of all the edges in WXG η emanating from component A.

Some SCTs input a CFG but actually use the number of elements flowing from one node to another, information typically encoded using a WXG. Instead of defining a new WXG we define and use a function Λ that transforms

a CFG into a CXG, i.e. $\Lambda : CFG \rightarrow WXG$.

$$\mathbf{\Lambda}\psi(s_1, s_2) = |\psi(s_1, _, s_2)|$$

Our use of this transformation does not necessarily imply that any implementation of that SCT will actually require that the transformation be computed. Using a transformation helps in differentiating the type of relation input by a SCT and the manner in which it is used. The knowledge that a SCT actually uses a WXG but is described to input a CFG is preserved in our reformulation of the SCTs.

3.1.2 On global variables and types

Figures 1, 2, and 3 define the majority of the interconnection relations unambiguously, except those involving global variables and types. What constitutes a global variable or a type and what it means to use, modify, or reference such a component differs between SCTs. In several instances these phrases and terms have only been defined loosely, most often by examples. In the absence of precise definitions in an original work, any precise reformulation of it has the potential of misrepresenting the work. Hence, we do not attempt to give precise meanings to these phrases, but instead present a summary of their usage below.

The definition of a "global variable" tends to be influenced by the programming language the SCT is designed for and/or experimented with. In FORTRAN a variable appearing in a COMMON block is global [28]. For C, some SCTs define variables "`extern`," i.e., external to a function, to be global [27]. Others consider local variables that are defined as "`static`," i.e., within the local scope of a function but with global lifetime, as global variables too [21].

The definition of what constitutes the use or modification of a variable also becomes difficult when a program contains pointers. Since the SCTs we describe do not make these terms any more precise, we refrain from attempting a better definition. A variable is said to be "referenced" if it is "used" or "modified."

While the SCTs use the term "global variable" within the parameters of the conventionally accepted usage, the same is not true about the terms "type." Consider the following code fragment in C:

```
typedef Storage char[];
    struct Stack {
            int sp;
            Storage buffer;
    } S;
```

According to the conventional definition, this code fragment *introduces* the types `Storage` and `Stack`.

However, Patel *et al.* also consider a field of a record (i.e., a `struct` in C) to be a type [31]. Thus, according to Patel *et al.* the above fragment *also* introduces types `Stack.sp` and `Stack.buffer`.

Definition: A type (say x) is a *sub-part* of another type (say y) if x is used in defining y. Conversely, type y is said to be a *super-part* of x. The sub-part and super-part relations are transitive, asymmetric, and irreflexive, and hence define a partial order [31].

In the above example, by conventional definition, `Storage` and `int` are sub-parts of `Stack`. However, according to Patel *et al.*'s definition, `Storage.buffer` and `Storage.sp` too are sub-parts of `Stack`.

Definition: A procedure p *directly references* a type t if it references a variable, say v, of type t. A procedure *downward references* the sub-parts of types it directly references. It *upward references* the super-parts of types it directly references. (The phrases "upward" and "downward" imply directions in a tree representation of the sub-part relation.)

Patel *et al.*'s SCT uses a WXG η_t that counts the direct and upward references to types in a procedure. For the above example, the expression `S.sp` will be counted as a reference to each of the types `Stack`, `Stack.sp`, and `int`. Ogando *et al.* use a CXG π_{ti} that maintains the direct and downward references to types in the interface specification of a procedure [27]. In this CXG, a formal parameter of type `struct Stack`, as in our example, also defines upward references to types `int`, `Storage`, and `char`.

Patel *et al.* and Ogando *et al.* use the terms *sub-type* and *super-type*, respectively, to mean *sub-part* and *super-part*. We use the latter terms because the former have a well accepted meaning in the context of type inheritance.

3.1.3 Filtering interconnection graphs

Most SCTs filter the input interconnection graphs for "noise" that may negatively influence the subsystems they recover. We consider such operations separate from the computations performed to create subsystems. The filters applied by various SCTs may be summarized as follows:

1. Remove a relation between a procedure and a type (say t) if there exists a relation between that procedure and a super-part of type t [21, 27].
2. Remove "utility" components from an interconnection graph. These are components (usually procedures and global variables) connected to several other components [6, 23]. Functions and procedures in an external library or in a user-defined library are typical examples of utility components. Hutchens and Basili also consider routines that "do all their communication via parameters and return values and are also called by more than one routine" as utility components [17].
3. Combine (not remove) utility components belonging to a standard library into a single component [25].
4. Remove (rather, do not include) information pertaining to "loop variables" from the interconnection graph [31].
5. Remove all lexical affinities whose "resolving power" (see Section 4.1.2) in a document is one standard deviation above the mean resolving power of all the lexical affinities in that document [22].
6. Remove components connected to single components [36].

3.2 Properties of a subsystem classification

The subsystem classification recovered by a SCT may be described in terms of:

1. the set of program components included in it,
2. its organizational structure of its subsystems, and
3. the interpretation assigned to the structure.

These issues are discussed in the rest of this section.

3.2.1 Program components included in subsystem classification

Let $\mathcal{T}$ be the set of program components in the subsystems recovered by a SCT. Clearly, for any SCT, $\mathcal{T}$ would be a subset of the set $\mathcal{S}$, the set of all program components used in the interconnection relations it inputs. For some SCTs the set $\mathcal{T}$ may be constrained further to $\mathcal{P}$ (the set of source nodes) or $\mathcal{P} \cup \mathcal{Q}$ (the set of source and target nodes), see Table 4.

3.2.2 Organizational structure of a subsystem classification

In the simplest case, each subsystem may simply be a *subset* of the program components. Such a subsystem classification is termed *flat*. If the pairwise intersection of all of the subsystems of a flat subsystem classification is empty, the subsystem classification is termed *partitioned*.

When attempting to recover modules, each subsystem in a partitioned subsystem classification may be interpreted as representing a module. Since a program component in a partitioned subsystem classification belongs to only one subsystem, a component is associated with only one module. This is what is usually desired.

Given that all the SCTs are essentially heuristic, requiring that a subsystem classification be partitioned is too strong a constraint. Some SCTs create classifications that are flat, but not partitioned. There are also SCTs that create classifications that are not even flat. A subsystem classification is not flat when its subsystems may contain other subsystems, not just program components. Such subsystem classifications are termed *stratified* if they can be represented by directed acyclic graphs, with program components at the terminal nodes, intermediate nodes representing subsystems, and edges going from the node of a subsystem to those of its members. It is sometimes desired that a subsystem classification be a tree, not just a directed acyclic graph, implying that each subsystem or program component belongs to at most one subsystem. A stratified subsystem classification that is also a tree is called *hierarchic*.

There is a special type of hierarchic subsystem classification called a *dendrogram* which is recovered by SCTs that use numerical cluster analysis [19]. A dendrogram may be visualized as a tree with numeric levels assigned to each intermediate node, such that the numeric levels increase monotonically from a child node to the parent node. This may be represented diagrammatically by drawing a tree such that the length of the line connecting a parent

and a child subsystem is proportional to the difference in their levels. An example dendrogram is shown in Figure 1; the ordinate axis in the figure shows the absolute level numbers.

The level numbers in a dendrogram may be used to define an ordering "immediately below" between pairs of nodes even if they are not in the same subtree.

Definition: A node or a leaf Y is *immediately below* a node X if (a) Y is at a level less than or equal to X and (b) the parent node of Y is *not* immediately below X. Further, a node is said to be *immediately below* itself.

In the diagrammatic representation of a dendrogram, stated earlier, each branch cut by a line drawn through a node X and perpendicular to the ordinate axis identifies a node or leaf immediately below X. In Figure 1 node M_1 and M_3 are immediately below M_3; D, G, M_2, and M_1 are immediately below M_1.

3.2.3 Interpretations of an organization structure

A subsystem consists of components that are related to each other. One can arrive at different subsystem classification based on the specific relationships used to group components. Some of the relationships that SCTs use to relate components are: (a) the components belong to the same module, (b) the components are affected by the same change request, (c) the components are related to the same software error. What it means for a pair of program components to be in the same subsystem therefore depends on the SCT.

A fundamental question that remains unanswered is "What constitutes a subsystem?" A subsystem is clearly a subset of program components for flat subsystem classifications. The definition of a subsystem is not quite so straightforward when subsystem classifications are stratified.

Consider the tree in Figure 1. It represents the output of SCTs generating hierarchical subsystem classifications. The symbols $A \ldots G$ represent program components and the symbols $M_1 \ldots M_4$ represent intermediate nodes of the tree. Clearly, each non-leaf node in the tree represents a subsystem. The question is "Who are its members?"

There are three interpretations commonly assigned to a hierarchic subsystem classification, each giving a different set of members in the subsystems associated with the nodes of the tree. We use the symbols I_1, I_2, and I_3 such that $I_k(M_j)$ gives the members of subsystem M_j using interpretation k.

Interpretation 1: A subsystem with respect to an intermediate node of a hierarchic subsystem classification consists of its children. Thus, for Figure 1:

$$\begin{aligned} I_1(M_1) &= \{A, B, C\} \\ I_1(M_2) &= \{E, F\} \\ I_1(M_3) &= \{D, G, M_2\} \\ I_1(M_4) &= \{M_1, M_3\} \end{aligned}$$

According to this interpretation, a subsystem is a set consisting of program components and other subsystems [11, 25].

Interpretation 2: The program components at the leaves of a subtree rooted at a node belong to the subsystem represented by that node. Thus, for Figure 1:

$$\begin{aligned} I_2(M_1) &= \{A, B, C\} \\ I_2(M_2) &= \{E, F\} \\ I_2(M_3) &= \{D, E, F, G\} \\ I_2(M_4) &= \{A, B, C, D, E, F, G\} \end{aligned}$$

This interpretation creates subsystems that are sets consisting only of program components. The resulting subsystems may overlap, but must comply with the constraint that if the intersection of two subsystems is non-empty, then one of them is a proper subset of the other [23].

Interpretation 3: The third interpretation is usually assigned to dendrograms. The subsystem at each node of a dendrogram consists of a partition of all the program components in the dendrogram, such that if node X is immediately below node Y, then the *set* of program components in the subtree rooted at X is a *subset in* the

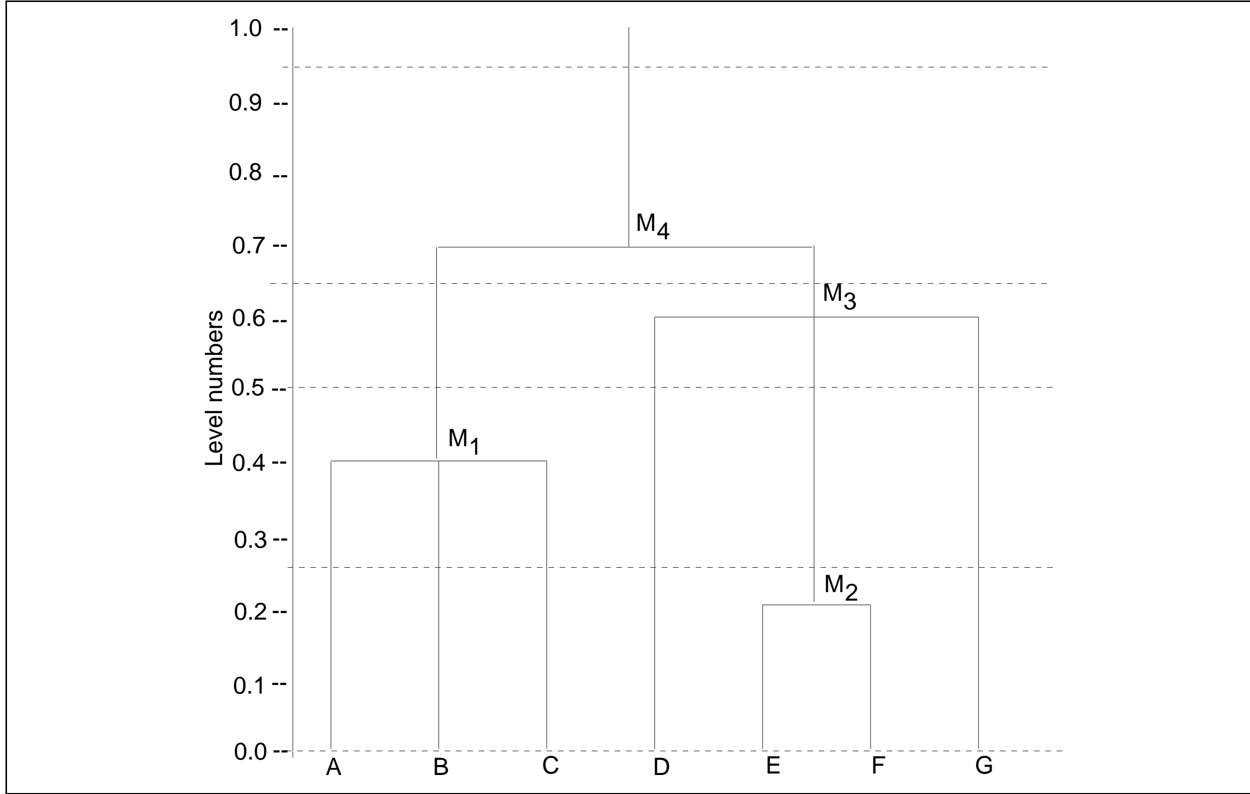

Figure 1 A sample hierarchical subsystem classification.

partition of Y, and vice-versa. In other words, $I_2(X) \in I_3(Y)$ if and only if X is immediately below Y. Thus for Figure 1:

$$\begin{aligned}
L(M_1) = 0.4;\ \ & I_3(M_1) = \{\{A, B, C\}, \{E, F\}, \{D\}, \{G\}\} \\
L(M_2) = 0.2;\ \ & I_3(M_2) = \{\{A\}, \{B\}, \{C\}, \{E, F\}, \{D\}, \{G\}\} \\
L(M_3) = 0.6;\ \ & I_3(M_3) = \{\{A, B, C\}, \{D, E, F, G\}\} \\
L(M_4) = 0.7;\ \ & I_3(M_4) = \{\{A, B, C, D, E, F, G\}\}
\end{aligned}$$

where $L(M_i)$ gives the level of subsystem M_i in the dendrogram.

A dendrogram, therefore, represents layers of partitioned subsystem classifications, with a level number associated to each layer. The partition at the highest level places all program components in one set. The partition at the lowest level places each component in a subsystem by itself. Further, all components placed in the same subsystem at one level are also placed in the same subsystems at every higher level. The above interpretation of a dendrogram follows strictly from its definition [19] and is used by SCTs employing numerical clustering techniques.

3.3 Computation performed by a SCT

Depending upon the type of computation (algorithms) they use, SCTs may largely be classified as *graph-theoretic*, *conceptual*, *numerical*, or *flow-analysis based*, as follows:

1. If a technique uses graph-theoretic computations, it is termed *graph-theoretic*.
2. If a technique uses computations that may be explained using flow-analysis concepts [3], it is termed *flow-analysis based*.

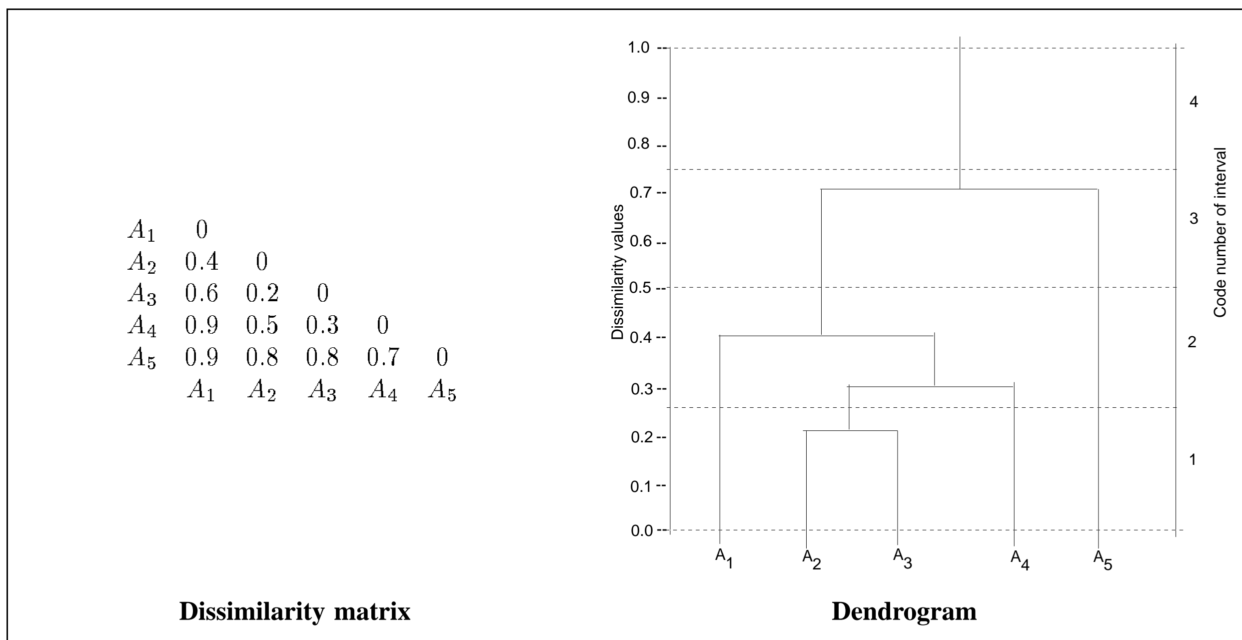

Figure 2 A sample dissimilarity matrix and the corresponding dendrogram resulting from single-link HAC. Since the matrix is symmetric, only the lower triangle and the diagonal are shown. A dendrogram is a hierarchy of a set of equivalence relations (i.e., partitions). It may be represented as a tree. 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 components in the subtree rooted at that branch. Components in a partition at a lower dissimilarity value are more likely to be similar. At the highest level of dissimilarity value all components 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.

3. If the values of attributes are measured on an ordinal scale, the technique is termed *conceptual* [24].
4. However, if attribute values are measured on a ratio or an interval scale and numeric computations are used, the technique is termed *numeric*.

SCTs generating stratified subsystem classifications may also be classified as *top-down* or *bottom-up* [23]. The top-down SCTs create subsystems at the top of a stratified subsystem classification and then iteratively *decompose* them to create subsystems at lower levels. Bottom-up SCTs create subsystems at the bottom of a subsystem classification and then iteratively *merge* subsystems to create those at the higher levels.

All stratified SCTs studied in this paper, except Choi and Scacchi's [11], are bottom-up. The computation of top-down stratified SCTs may be abstracted by the algorithm template of Figure 3. Starting with an interconnection graph, these algorithms iteratively find a set of components that may be said to be most similar (Step *s1*), place these components into the same subsystem (Step *s2*), and create a new interconnection graph by combining the two components (Step *s3*). Stratified SCTs differ in the strategies chosen at each of these steps.

Notice that a subsystem classification created by these algorithms consists of program components forming the source and target nodes ($\mathcal{P} \cup \mathcal{Q}$) of the interconnection graph it inputs. However, Table 4, Column "Elements from," shows that some of the stratified SCTs organize a different set of program components. This is because most SCTs transform their input interconnection graph before stratifying it. Table 4, Column "Elements from," identifies the set with respect to the input to a SCT, not its stratification algorithm.

An organization of a set of entities into subsets such that entities in the same subset are in "some sense" more similar to each other than to those in different subsets is often referred to as a *cluster* [15]. Since a subsystem classification is analogous to a cluster [23], some of the SCTs we study use *hierarchical agglomerative clustering*, or HAC [19]—a class of numerical cluster analysis algorithms—or its variations.

```
input:  An interconnection graph d.
output:  A stratified subsystem classification over the set 𝒫 ∪ 𝒬
s0:  put each element of 𝒫 ∪ 𝒬 in a subsystem by itself
while d has more than one element do

    s1:  identify a set (with at least two elements) of most similar subsystems
    s2:  create a new subsystem by merging the similar subsystems
    s3:  create a new d by replacing the similar subsystems by the new subsystem

end-while
```

Figure 3 Algorithm template for generating stratified SCTs by top-down computation. A specific stratified SCT may be described by describing the input interconnection graph it inputs and the strategies it uses at each of the Steps *s1*, *s2*, and *s3*.

The choices at Steps *s1*, *s2*, and *s3* may be restricted further if a SCT uses HAC. The input to a HAC algorithm is a symmetric, directed WXG without self edges (i.e., there is no edge from a node to itself). The strategy at Step *s1* depends upon whether the weights on the WXG represent (a) similarity between two nodes or (b) dissimilarity between two nodes. Two nodes are *most similar* if they have either the highest similarity or the lowest dissimilarity. Step *s2* of a HAC algorithm identifies a set of nodes that are pairwise most similar.

Since HACs output dendrograms, the strategy used at Step *s2* must satisfy interpretation I_3, defined earlier. This may be achieved by creating a new cluster by taking the union of the most similar clusters. The (dis)similarity value between the clusters merged is assigned as the level number for the new cluster.

HAC algorithms traditionally choose from one of four strategies [19]—*single-link*, *complete-link*, *weighted-average-link*, and *unweighted-average-link*—to create a new WXG after merging a set of similar clusters to create a new cluster (Figure 3, Step *s3*). Let X be the set of similar clusters being combined in a given iteration, n be the new cluster replacing all the members of X, and e be another cluster (not in X). In the single-link strategy the smallest dissimilarity between any element of X and e is used as the dissimilarity between n and e. The complete-link strategy uses the largest such dissimilarity and the other two strategies use weighted and unweighted averages of dissimilarities between e and members of X. Some SCTs use yet another strategy, termed *cumulative-link*, where the sum of the dissimilarities between e and all the elements of X is taken as the dissimilarity between e and n.

The dendrogram in Figure 2 is the result of applying 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 components form a single cluster at level 0.7.

4 Reformulation of SCTs using our unified framework

This section gives an overview of various of SCTs. 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 4 classifies SCTs studied in this paper based on the classification scheme developed in Section 3. The details of the computations are elaborated upon below.

4.1 Numeric, stratified SCTs

The numeric, stratified SCTs we examine either use HAC or some variation of HAC. To describe a SCT that uses a HAC algorithm, it is sufficient to describe (a) whether it uses a similarity matrix or a dissimilarity matrix, (b) how this matrix is computed, and (c) which of the five strategies—*single-link*, *complete-link*, *weighted-average-link*, *unweighted-average-link*, or *cumulative-link*—it uses at Step *s3*.

We use the symbols $diss_i$ and sim_j to denote a dissimilarity matrix and a similarity matrix, respectively.

Table 4 Classification of SCTs based on criteria introduced in Section 3.

SCT of ..	Computation used	Input graph	Subsystem classification characteristics		
			Elements from	Partitioned?	Stratified? (not flat?)
Belady & Evangelisti, 81	numeric	π_{gr}	$\mathcal{S}$	yes	no
Choi & Scacchi, 90	graph-theoretic	ψ_F	$\mathcal{P} \cup \mathcal{Q}$	yes	yes
Hutchens & Basili, 85 (both ARTs)	numeric	ψ_{gx}	$\mathcal{P} \cup \mathcal{Q}$	yes	yes
Livadas & Johnson, 94	graph-theoretic	$\pi_{gi}, \pi_{tg}, \pi_{tl}, \pi_{ti}$	$\mathcal{S}$	no	no
Maarek *et. al.*, 91	numeric	η_w	$\mathcal{P}$	yes	yes
Maarek & Kaiser, 88	numeric	ψ_F	$\mathcal{P} \cup \mathcal{Q}$	yes	yes
Müller & Uhl, 90	mixed	Generic π, ψ, η	$\mathcal{P}$	no	yes
Ogando *et al.*, 94	graph-theoretic	π_{gr}, π_{ti}	$\mathcal{S}$	no	no
Ong & Tsai, 93	graph-theoretic	$\pi_{def}, \pi_{live}, \pi_{set}, \pi_{use}, \ldots$	$\mathcal{S}$	no	no
Patel *et. al.*, 92	numeric	η_t	$\mathcal{P}$	no	no
Schwanke, 91	numeric	π_{su}, π_c	$\mathcal{P}$	yes	yes
Selby & Basili, 91	numeric	ψ_{gx}	$\mathcal{P} \cup \mathcal{Q}$	yes	yes

4.1.1 SCTs of Hutchens and Basili

Hutchens and Basili suggest two SCTs [17]. Both use CFG ψ_{gx} and create subsystems of components from the corresponding set $\mathcal{P} \cup \mathcal{Q}$. The relation $\psi_{gx}(A, x, B)$ implies that procedure A refers to a global variable x that is modified by procedure B. Even though CFG ψ_{gx} may be computed from CFGs π_{gr} and π_{gu}, $\psi_{gx}(s_1, x, s_2) = \pi_{gr}(s_1, x) \wedge \pi_{gu}(sx, s_2)$, we maintain that this SCT inputs a CFG, not two CXGs. This is because Hutchens and Basili also investigate other CFGs ψ_{gr} and ψ_{gc}, but choose CFG ψ_{gx} because it is computationally less expensive to construct.

Both of the SCTs of Hutchens and Basili use dissimilarity matrices but each uses a different formula to compute it. The first SCT computes the dissimilarity matrix, termed *recomputed dissimilarity*, as follows:

$$diss_1(A, B) = \frac{degree(A) + degree(B) - 2sim_2(A, B)}{degree(A) + degree(B) - sim_2(A, B)}$$

where matrix sim_2 gives the *binding strength* between two procedures and the vector $degree$ gives the number of bindings involving a given procedure, as follows:

$$sim_2(A, B) = \Lambda\psi_{gx}(A, B) + \Lambda\psi_{gx}(B, A)$$

$$degree(A) = \sum \Lambda\psi_{gx}(A, _) + \sum \Lambda\psi_{gx}(_, A)$$

The second SCT computes the *expected dissimilarity* using the formula:

$$diss_3(A, B) = \frac{degree(A) + degree(B)}{(\dim(sim_2) - 1) \times sim_2(A, B)}$$

where sim_2 and $degree$ are as defined for recomputed dissimilarity and $\dim(sim_2)$ gives the dimension of sim_2, i.e., its number of rows or number columns (the two are same since a dissimilarity matrix is square).

The SCTs of Hutchens and Basili use the stratification algorithm template of Figure 3, generate dendrograms, and use the cumulative-link strategy to compute the coefficients for the new element that replaces the pair of elements combined into a subsystem. However, they are not HAC algorithms because they maintain two matrices in each iteration—a similarity matrix (sim_2) and a dissimilarity matrix ($diss_1$ or $diss_2$)—instead of just one. In Step *s1*, the two algorithms first compute the respective dissimilarity matrix from the similarity matrix sim_2 and they then use the dissimilarity matrix to find the pair of similar elements. In Step *s3*, they compute a new similarity matrix sim_2 using the cumulative-link strategy. Thus, instead of doing all the operations on either a similarity matrix or a dissimilarity matrix, as is done by a HAC algorithm, these SCTs compute two matrices in each iteration.

The recomputation of a dissimilarity matrix from a similarity matrix in each iteration leads to the possibility that the dissimilarity values over successive iterations may decrease, implying that an iteration may create a subsystem with a lower dissimilarity than a subsystem created in an earlier iteration. The hierarchy of partitions thus created may have decreasing level numbers. The resulting classification, therefore, will not be a dendrogram. To overcome this problem, Hutchens and Basili suggest that whenever the dissimilarity of a newly created subsystem is smaller than the subsystem created in the previous iteration, the former should be assigned the same dissimilarity level as the latter.

4.1.2 SCT of Maarek *et al.*

Maarek *et al.*'s SCT differs from other numeric, stratified techniques in that it uses information from program documentation, not the code itself [22]. It uses WXG η_w, relating the documentation of a program and its lexical affinities, and classifies symbols belonging to the corresponding set $\mathcal{P}$ into subsystems. Maarek *et al.* suggest using either single-link or complete-link HAC with the dissimilarity matrix created by the following function:

$$diss_4(A,B) = \sum_{i \in a_A \cap a_B} \hat{\rho}_A(i) \times \hat{\rho}_B(i)$$

where:

- $a_X = \{F : \eta_w(X,F)\}$, i.e., the set of lexical affinities found in a document X,
- $\rho_X(F) = -\eta_w(X,F) \times \log(Pr(F,X))$, i.e., resolving power of lexical affinity F in a document X,
- $Pr(X,F) = \eta_w(F,X)/\sum \eta_w(F,_)$, i.e., probability that lexical affinity F appears in the documentation of program X,
- $\hat{\rho}_X(F) = (\rho_X(F) - \bar{\rho_X})/\sigma_{\rho_X}$, i.e., normalized resolving power of lexical affinity F in document X, and
- $\bar{\rho_X}$ is the mean of ρ_X and σ_{ρ_X} is its standard deviation.

4.1.3 SCT of Maarek and Kaiser

Maarek and Kaiser's SCT uses CFG ψ_F (that relates a file declaring a syntactic-unit to a file using that unit) and creates subsystems consisting of symbols from the corresponding set $\mathcal{P} \cup \mathcal{Q}$ [23]. This SCT is an extension of single-link HAC. It uses the similarity matrix created from WXG $\Lambda\psi_F$.

$$sim_5(A,B) = \Lambda\psi_F(A,B) + \Lambda\psi_F(B,A)$$

Maarek and Kaiser's SCT extends the single-link HAC in that it marks the partitions created at each level of the dendrogram as either *frozen* or *prospective*. The final subsystem classification consists only of the partitions marked frozen. A partition is marked frozen when v_α, the variance of the size of each subset in the partition from an *example* size α is small. If there are k subsets $x_1, x_2, \ldots, x_n$ in partition p, then

$$v_\alpha(p) = \frac{1}{k}\sum_{i=1}^{k}([x_i] - \alpha)^2$$

where the expression $[x_i]$ denotes the size of x_i, defined as follows. If partition p is marked frozen then the size of its subsets is forced to be 1, i.e., $[x_i] = 1$, $1 \leq i \leq k$. If partition p is prospective then $[x_i] = |x_i|$.

To determine a partition with a 'small' v_α, the v_α of up to n consecutive prospective partitions generated by the single-link HAC are compared. The one with the smallest v_α is considered 'small' and is frozen. When $n = 1$, all partitions are marked frozen, thus resulting in a single-link HAC. When $n = 2$, the v_α's of at most two consecutive partitions need be compared. This can be achieved by introducing the following step into the stratification algorithm:

```
s4:   if vα(new d) > vα(d)
         then mark d to be frozen
         else mark d to be prospective
```

where the (dis)similarity matrices d and new d also implicitly represent the partitions before and after the loop is executed. In addition, the partition represented by matrix d before entering the loop and after its termination are also marked as frozen.

4.1.4 SCT of Schwanke

Schwanke's SCT uses the CXGs π_c (which procedure calls which procedure) and π_{su} (which procedure references which syntactic unit) and creates subsystems consisting of components from the corresponding set $\mathcal{P}$ [36]. The similarity matrix it creates may be stated as:

$$sim_6(A, B) = \frac{w(r_A \cap r_B) + k \times (\pi_c(A, B) \vee \pi_c(B, A))}{n + w(r_A \cap r_B) + d \times (w(r_A - r_B) + w(r_B - r_A))}$$

where:

- n, k, and d are user defined parameters,
- $r_A = \{F : \pi_{su}(A, F)\}$, i.e., the set of syntactic units referenced 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 X of syntactic units, and
- $Pr(x) = |\pi_{su}(_, x)|/|\pi_{su}(_, _)|$, i.e., the proportion of use relations involving syntactic unit x (or the probability that a relation involves syntactic unit x).

Schwanke's SCT uses the single-link HAC algorithm for clustering program components. It also provides two interactive interfaces to validate a subsystem as it is created. In the first interface, after every subsystem is created the user is asked to confirm it. In the second interface, a new subsystem is first validated by using a heuristic; if the validation fails then the user is queried. Schwanke's heuristics may be stated as: If the syntactic units being clustered are declared in the same file, then the clustering is okay, else it is not. Implementation of this heuristic requires CXG π_d, i.e., which program element is declared in which file.

4.1.5 SCT of Selby and Basili

Selby and Basili's SCT uses the same CFG as the SCTs of Hutchens and Basili, Section 4.1.1, and creates subsystems consisting of symbols from the corresponding set $\mathcal{P}$ [37]. It uses single-link HAC with the similarity matrix sim_2 as defined with Hutchens and Basili's SCTs.

4.2 Numeric, non-stratified SCTs

Numeric, non-stratified SCTs are techniques that use numeric computation to organize program components into subsystems. They are non-stratified in that they only create one set of subsystems, not levels of subsystems. There are three SCTs that fall in this category. The subsystems produced by two of the techniques are partitioned [6, 1], but those produced by the remaining are not [31]. We have not had the opportunity to reformulate one of the SCTs since it appeared very recently [1]. A reformulation of the other SCTs is presented below.

4.2.1 SCT of Belady and Evangelisti

Belady and Evangelisti's SCT [6] uses the CXG π_{gr}, i.e., which function references which global variable*, and creates subsystems consisting of components from the corresponding set $\mathcal{S}$. It creates a similarity matrix which is simply the adjacency matrix representation of π_{gr}.

$$sim_7(A, B) = \pi_{gr}(A, B) \vee \pi_{gr}(B, A)$$

This similarity matrix is input to Donath and Hoffman's clustering algorithm to create subsystems consisting of components that are close to each other when placed in an N-dimensional space [13]. The placement of each component on an N-space is done by computing N (typically = 5) eigenvectors. The ith value of these vectors gives the coordinates of the ith element in the similarity matrix. The partitions created by Donath and Hoffman's algorithm are constrained by two parameters: the number of subsets to be generated and the maximum size of each subset. The details of the algorithm can be found in the original paper [13] .

4.2.2 SCT of Patel *et al.*

Patel *et al.*'s SCT uses η_t, the WXG representing the number of times a procedure has a direct or upward reference to a type [31]. It classifies symbols from the corresponding set $\mathcal{P}$ into subsystems. It creates a similarity matrix using the following function:

$$sim_8(A, B) = \frac{\vec{\eta_t}(A) \times \vec{\eta_t}(B)}{\|\vec{\eta_t}(A)\| \times \|\vec{\eta_t}(B)\|}$$

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

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

and the vector dimension represents the computation:

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

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

Patel *et al.*'s SCT does not generate a set of subsystems representing modules. Instead, it provides a function to test if a set of procedures belong to the same module. A set of procedures S constitutes a module if $T(S) \geq \tau$, where τ is some experimentally determined threshold value and $T(S)$ is defined as:

$$T(S) = \frac{\sum_{x,y \in S; x \neq y} sim_8(x, y)}{|S| \times |S - 1|}$$

4.3 Mixed, stratified SCTs

Mixed, stratified SCTs are techniques that use a combination of numeric and graph theoretic computations to organize program components into subsystems. They are stratified in that a subsystem recovered by these techniques may contain other subsystems, and such inclusion is free of cycles. Of the SCTs we study, only Müller and Uhl's may be classified as mixed, stratified.

---

* Belady and Evangelisti [6] 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 π_{gr} relation.

4.3.1 SCT of Müller and Uhl Müller and Uhl's SCT is programming language and context independent in that it does not commit to any specific type of input. It accepts any type of interconnection relations, i.e., either CXG, WXG, or CFG, but treats them all as WXGs [25]. Thus a CFG ψ input to Müller and Uhl's SCT is treated as a WXG $\Lambda\psi$, and a CXG π is treated as a WXG by associating a weight of one to each edge. We, therefore, use a generic WXG η to represent the interconnection relations input by this SCT.

Müller and Uhl's SCT is top-down and can be abstracted by the stratified SCT template of Figure 3, (i.e., iteratively select similar elements, combine them, and create a new interconnection graph). It differs from other stratified SCTs in the following ways.

1. During each iteration it computes several similarity matrices and provides a collection of rules to discern if two elements are similar. Other techniques usually compute only one (dis)similarity matrix and provide only one rule for detecting similar elements.
2. Besides deciding whether two elements are similar, it also provides rules to decide if two elements are *not* similar. Thus, it allows elements to be placed in *different* subsystems. Other SCTs only permit the selection of similar elements.
3. It merges similar elements in the interconnection graph, not the similarity matrix. The similarity matrices are recomputed from the new interconnection graph.
4. The subsystem classification created by this SCT, though stratified, may not be hierarchic.

Its rules for identifying similar elements may be classified as based on: (1) interconnection strength, (2) common neighbors, (3) centricity, and (4) name. The first three rules use similarity matrices like those used by HAC based SCTs. These computations are, however, defined 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:

$$sim_9(A, B) = \eta(A, B) + \eta(B, A)$$

Müller and 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 that are strongly related are placed in the same subsystem and those that are weakly related are placed in different subsystems.

Müller and Uhl suggest two similarity measures based on common neighbors, one using *common successors*:

$$sim_{10}(A, B) = |\eta(A, _) \cap \eta(B, _)|$$

and the other using *common predecessors*:

$$sim_{11}(A, B) = |\eta(_, A) \cap \eta(_, B)|$$

Two elements A and B are placed in the same subsystem either if $sim_{10}(A, B) \geq T_c$ or if $sim_{11}(A, B) \geq T_s$, where T_c and T_s are two threshold values.

Müller and Uhl define centricity as the number of dependences between an element and other elements:

$$degree(A) = \sum \eta(A, _) + \sum \eta(_, A)$$

This is the same as the degree of a weighted directed graph computed by adding the weights of its edges. Elements with $degree$ beyond thresholds T_k and T_f, are termed *central* and *fringe* elements, respectively. Fringe elements are assigned to different subsystems.

Identifying similar elements based on their names is unique to Müller and Uhl's SCT. Two program elements are considered to be similar if their names have matching substrings (e.g., common prefixes).

After identifying subsystems of similar elements, Müller and Uhl's SCT creates a new WXG using the cumulative-link strategy. Since this SCT allows multiple subsystems to be created during the same iteration, a component or a subsystem may be included in more than one subsystem leading to a non-hierarchical, yet stratified, subsystem classification. However, if during any iteration only one similarity criterion is applied, the subsystem classification would be hierarchical.

Müller and Uhl's SCT is interactive and leaves the selection of the appropriate rule(s) in each iteration to the user. Similarly, the choice of threshold values is also left to the user with the provision that these values may be changed between iterations.

4.4 Graph-theoretic, stratified SCTs

4.4.1 SCT of Choi and Scacchi Choi and Scacchi's SCT uses graph-theoretic computations and creates a stratified subsystem classification that is also a hierarchy [11]. This hierarchy is similar to that of a dendrogram, in that program components appear only at the leaf and that each component appears at most once. The intermediate nodes are abstract nodes representing subsystems. The hierarchy it generates is different from dendrograms in that there are no level numbers associated with the nodes.

Choi and Scacchi's SCT uses CFG ψ_F representing the flow of resources such as data types, procedures, and variables from one source code file to another. The subsystems it creates consist of symbols from the corresponding set $\mathcal{P} \cup \mathcal{Q}$. This SCT is top-down. It first finds the biconnected components and articulation points [2] of the graph. It then creates a subsystem for each articulation point of ψ_F. Each subsystem consists of an articulation point and sub-subsystems 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 approach extracts a subsystem classification with minimum alteration distance (sum of distances between pairs of leaves) and minimum coupling (sum of the number of children of all nodes).

Notice that this SCT does not make use of the tags on the edges of CFG ψ_F, therefore using it as a CXG.

4.5 Graph-theoretic, non-stratified SCTs

There are two SCTs that use graph-theoretic computations and generate non-stratified subsystems [21, 27]. Due to a close association between their developers the two SCTs have influenced each other and hence follow a similar theme. Both the SCTs aim at creating subsystems of procedures, types, and global variables representing modular subsystems and provide two strategies—*global-based* and *type-based*—to place program components in the same subsystem. The global-based strategy creates subsystems consisting of global variables and procedures whereas the type-based strategy creates subsystems consisting of types and procedures.

Before presenting the two SCTs, we introduce some additional definitions and notation to help us describe these SCTs. Figure 4 contains examples that enumerate the definitions introduced.

4.5.1 Additional definitions and notation The two SCTs use CXGs whose set of source nodes $\mathcal{P}$ and set of target nodes $\mathcal{Q}$ are disjoint, i.e., $\mathcal{P} \cap \mathcal{Q} = \phi$. The length of a path (sequence of edges traversed) in these graphs can be at most one. Hence, these algorithms employ operations that require traversing a single edge either *along* (i.e., from the source node to the target node) or *against* (i.e., from the target node to the source node).

Definition: The set of nodes reachable from a node a by traversing one edge of a CXG π are denoted as follows:

$$\pi^{+1}(x) = \{y | \pi(x,y) \vee \pi(y,x)\}$$
$$\pi(x) = \pi^1(x) = \pi^{+1}(x) \cup \{x\}$$

Whether a traversal is done along an edge or against an edge is obvious from the node itself, since a node in a CXG with disjoint $\mathcal{P}$ and $\mathcal{Q}$ can be either a target or a source of an edge, but not both, as enumerated by the example in Figure 4.

Definition: Two elements s_1 and s_2 have a *weak match* in CXG π if $\pi^{+1}(s_1) \cap \pi^{+1}(s_2) = \phi$. They have a *strong match* if $\pi^{+1}(s_1) = \pi^{+1}(s_2)$.

Definition: Let $WM(\pi)$ and $SM(\pi)$ denote the pairs of nodes in π with strong and weak match, respectively, i.e.:

$$(s_1, s_2) \in WM(\pi) \Leftrightarrow \pi^{+1}(s_1) \cap \pi^{+1}(s_2) \neq \phi$$
$$(s_1, s_2) \in SM(\pi) \Leftrightarrow \pi^{+1}(s_1) = \pi^{+1}(s_2)$$

Definition: Let $WM^*(\pi)$ give the reflexive, transitive-closure of $WM(\pi)$.

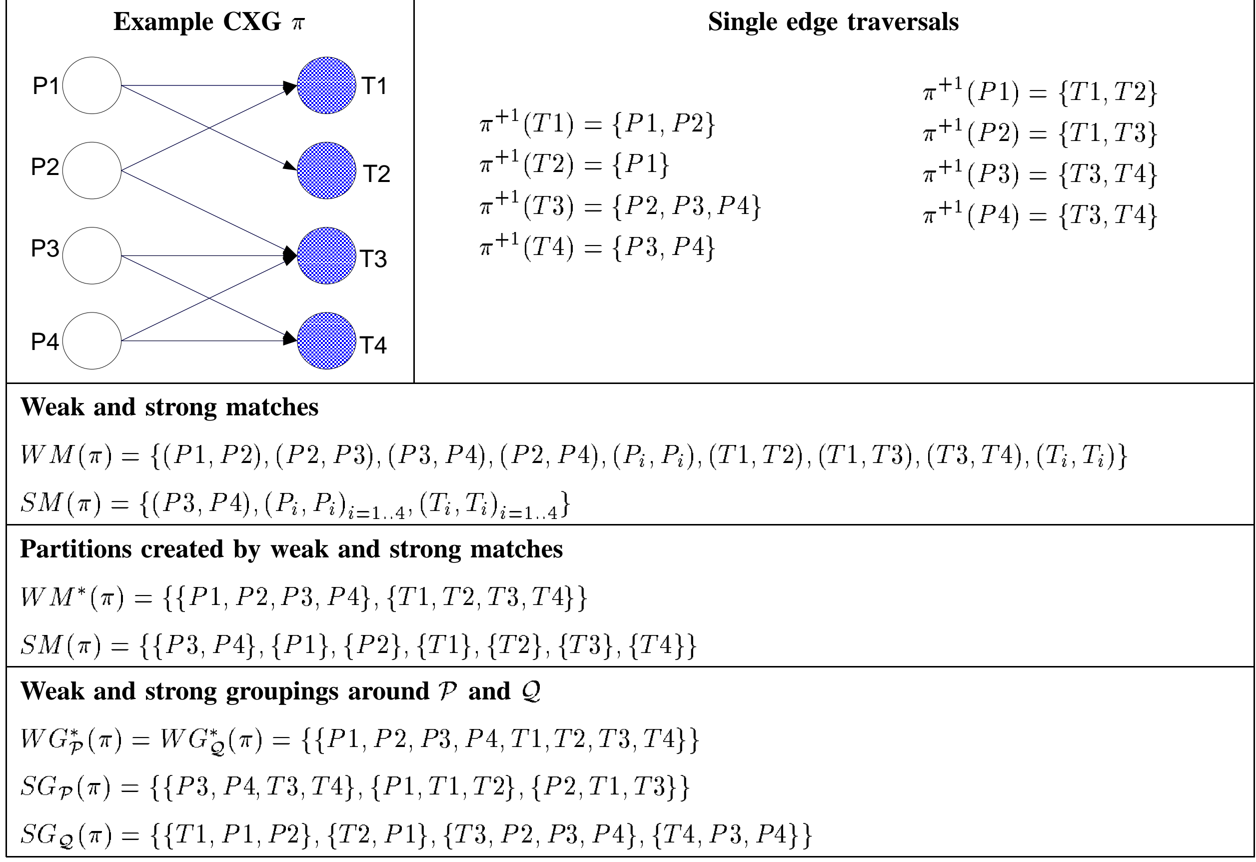

Figure 4 Examples enumerating definitions introduced to abstract and Livadas and Johnson's and Ogando *et al.*'s SCTs. These SCTs use CXGs with the property that $\mathcal{P} \cap \mathcal{Q} = \phi$

Definition: Let Z be a subset of $\mathcal{S}$. We use $WM^*_Z(\pi)$ and $SM_Z(\pi)$ to denote the following:

$$WM^*_Z(\pi) = WM^*(\pi) \cap Z \times Z$$
$$SM_Z(\pi) = SM(\pi) \cap Z \times Z$$

That is, $WM^*_Z(\pi)$ and $SM_Z(\pi)$ are subsets of $WM^*(\pi)$ and $SM(\pi)$, respectively, containing only relations between the elements of Z. If the sets $\mathcal{P}$ and $\mathcal{Q}$ of π are disjoint, then $WM^*_{\mathcal{P}}(\pi)$ and $WM^*_{\mathcal{Q}}(\pi)$ partition $WM^*(\pi)$, i.e.:

$$WM^*_{\mathcal{P}}(\pi) \cup WM^*_{\mathcal{Q}}(\pi) = WM^*(\pi)$$
$$WM^*_{\mathcal{P}}(\pi) \cap WM^*_{\mathcal{Q}}(\pi) = \phi$$

Similarly, $SM_{\mathcal{P}}(\pi)$ and $SM_{\mathcal{Q}}(\pi)$ partition $SM(\pi)$.

$WM^*(\pi)$ and $SM(\pi)$ are equivalence relations, hence they define partitions over the set $\mathcal{S}$. If the sets $\mathcal{P}$ and $\mathcal{Q}$ are disjoint, then $WM^*_{\mathcal{P}}(\pi)$ and $WM^*_{\mathcal{Q}}(\pi)$ (similarly, $SM_{\mathcal{P}}(\pi)$ and $SM_{\mathcal{Q}}(\pi)$) are also equivalence relations creating partitions over the sets $\mathcal{P}$ and $\mathcal{Q}$, respectively.

The subsets in the partitions created by weak and strong matches may be expanded further to create *weak groupings* (WG) and *strong groupings* (SG).

Definition: A *weak grouping* around $\mathcal{P}$, $WG^*_{\mathcal{P}}(\pi)$ is the set of subsets of $\mathcal{S}$ created by expanding the subsets in the partitions created by $WM^*_{\mathcal{P}}(\pi)$ as follows:

If D is a subset in the partition due to $WM^*_{\mathcal{P}}(\pi)$ then $D \cup \bigcup_{d \in D} \pi^{+1}(d) \in WM^*_{\mathcal{P}}(\pi)$.

A weak grouping around $\mathcal{Q}$, $WG^*_{\mathcal{Q}}(\pi)$; a strong grouping around $\mathcal{P}$, $SG_{\mathcal{P}}(\pi)$; and a strong grouping around $\mathcal{Q}$, $SG_{\mathcal{Q}}(\pi)$, may be similarly created by expanding the partitions created by $WM^*_{\mathcal{Q}}(\pi)$, $SM_{\mathcal{P}}(\pi)$, and $SM_{\mathcal{P}}(\pi)$, respectively.

4.5.2 SCT of Ogando *et al.*

The global-based strategy of Ogando *et al.* creates subsystems given by $WG^*_{\mathcal{Q}}(\pi_{gr})$, i.e., weak grouping around $\mathcal{Q}$ of the CXG π_{gr} (which procedure uses which global variable) [27]. The subsystems created by this strategy have the following properties:

1. If a procedure refers to the global variables g_1 and g_2 then the global variables are placed in the same subsystem.
2. If a global variable is placed in a subsystem then all procedures referring it are placed in that subsystem.
3. A global variable is placed in at most one subsystem.

Their type-based strategy uses CXG π_{ti} that represents relations between procedures and types used in its interface description (i.e., formal parameters and return value). This CXG maintains direct and downward references to types. The CXG is filtered to remove relations between a procedure p and a type t if procedure p references a super-part of t. The type-based strategy creates subsystems given by $WG^*_{\mathcal{Q}}(\pi_{ti})$, i.e., weak grouping around $\mathcal{Q}$ of CXG π_{ti} (after filtering, see Section 3.1.3). The subsystems created by this strategy may be described as follows:

1. If the interface of a procedure refers to types t_1 and t_2 but does not refer to any of their super-parts, then the two types are placed in the same subsystem.
2. If a type is placed in a subsystem, then all procedures referring to it (in the interface) are placed in that subsystem.
3. A type is placed in at most one subsystem.

4.5.3 SCTs of Livadas and Johnson

Livadas and Johnson's *global-based* strategy is the same as Ogando *et al.*'s, except that it uses the CXG π_{gi} instead of the CXG π_{gr} [21]. In other words, it creates subsystems given by $WG^*_{\mathcal{Q}}(\pi_{gi})$, i.e., weak grouping around $\mathcal{Q}$ of the CXG π_{gi} (which procedure references which global variable either directly or indirectly due to aliasing with a formal parameter).

Their type-based strategy is really a family of strategies in that it is parameterized by a CXG that gives a relation between procedures and types (i.e., which procedure uses which types). It creates subsystems given by $SG_{\mathcal{P}}(\pi)$, i.e., strong grouping around $\mathcal{P}$ of the CXG π. Livadas and Johnson have experimented with various combinations (unions) of CXGs π_{ti}, π_{tl}, and π_{tg}.

4.6 Flow-analysis based, non-stratified SCT

SCTs whose rationale is best expressed using flow-analysis terminology [3] are classified as flow-analysis based (even though flow-analysis may be stated in terms of graph operations). There are two such SCTs [28, 39]. These two SCTs have evolved from the respective author's experiences with reengineering procedural programs to object-oriented programs. The presentation of Silva-Lepe's technique primarily consists of a combination of examples and discussions [39]. This makes it harder to abstract it and hence we have not reformulated it. The second SCT is presented using some algorithms, besides examples and discussions [28]. Our reformulation of this SCT below, presents it with a precision greater than that in the original work.

4.6.1 SCT of Ong and Tsai

Ong and Tsai's SCT is designed to recover objects, i.e., modular subsystems, from Fortran programs [28]. The subsystems it recovers are flat and may have overlapping subsystems. Each subsystem recovered by this SCT consists of a set of variables and statement blocks. Therefore, Ong and Tsai's SCT does not treat a procedure or function as a unit that cannot be decomposed, as other SCTs do.

This SCT is similar to the graph-theoretic, non-stratified SCTs in that the source nodes $\mathcal{P}$ and target nodes $\mathcal{Q}$ of the CXGs it uses are disjoint, i.e., $\mathcal{P} \cap \mathcal{Q} = \phi$. Hence, we find it convenient to use the notation π^{+1} introduced in the previous subsection.

Ong and Tsai's SCT uses CXGs π_{use}, π_{set}, π_{def}, and π_{live}. Computing these CXGs requires static flow-analysis of the programs [3].

Ong and Tsai's SCT consists of two steps. In the first step it creates subsystems consisting of global variables, formal parameters, and actual parameters. The variables in each of these subsystems are said to identify instance variables of an object class. In the next step, statement blocks, signifying "methods" of the class, are added to each subsystem.

The placement of variables into subsystems, in the first step, is done using the following rules:

1. All variables belonging to the same COMMON block belong to the same subsystem.
2. The formal parameters of each procedure are split into three categories: *use-only*, *define-and-use*, and *define-only*. For each procedure, the set with the largest size forms a subsystem.
3. If a formal parameter of a procedure belongs to a subsystem, then the corresponding actual parameter of a call to this procedure also belongs to that subsystem.
4. If an actual parameter of a procedure call belongs to a subsystem, then the corresponding formal parameter of the procedure it calls also belongs to that subsystem.

(Note that this step uses interconnection information not contained in the CXGs listed in Figure 1.)

The subsystems of variables formed above are expanded to include statement blocks. Let V be the set of variables placed in subsystem g. Statement blocks are added to g using the following steps:

Let $S(V) = \bigcup_{v \in V} \left(\pi^{+1}_{def}(v) \cup \pi^{+1}_{use}(v) \right)$. The set of statements that define or use a variable in V.

For every statement s in $S(V)$ **do**

For every statement-block b containing statement s, starting from the smallest block and going to bigger blocks, **do**

- $FP(b) = \pi^{+1}_{use}(b) - V$. The set of variables used in block b, except those in V. If block b was used to make a method, then these variables would correspond to the formal parameters.
- $RV(b) = \pi^{+1}_{set}(b) \cap \pi^{+1}_{live}(b) - D$. The set of variables, other than those in V, that are used outside block b after being modified in block b. If block b was used to make a method, then these variables would be used to return values.

If the sets $FP(b)$ and $RV(b)$ are *small* **then** include block b in subsystem g

Whether the sets $FP(b)$ and $RV(b)$ are small is determined "by comparing several FP's and RV's for a gradually expanding b that consists of the statements around s [28]." Since FP and RV are not constrained to be "smallest," more than one statement block b containing a statement s may qualify to be small and be included in the subsystem. This allows placement of overlapping statement blocks in the same subsystem. Similarly, a statement block may also be placed in multiple subsystems. Thus, Ong and Tsai's SCT generates non-partitioned subsystems.

5 Benefits of our unified model

Our unified model offers the following important benefits.

- It identifies the parameters that are important to describe SCTs. If the SCTs are expressed using the terminology introduced, it also makes it easier to compare various SCTs.
- Separating the computation performed by the SCTs from their inputs paves the way to create new SCTs by mixing and matching the information used and the computations performed by different SCTs.

Additionally, a comprehensive survey of the interconnection relations used by SCTs, Figures 1, 2, 3, and 4, may also be used to:

- Identify SCTs that may be suitable in a particular context.
- Identify "requirements" (or develop benchmarks) for language processing tools when developing specific SCTs.

These benefits are further elaborated upon below.

5.1 Describing and comparing SCTs

We have developed a classification of the inputs, outputs, and computations performed by SCTs. New SCTs may be expressed concisely if they can be described using our classification. For instance, if a new SCT generates stratified subsystems, one may state whether it may be abstracted using the template in Figure 3, and if so, then the decisions made in Steps *s1*, *s2*, and *s3* may be explained. If the stratified SCT happens to be a HAC, then only the formula to compute its similarity matrix or dissimilarity matrix need be stated.

Most SCTs, except those influenced by numerical cluster analysis [19], have been described in the original works using different terminology and notation. Reformulating them using our unified framework highlights the similarities and differences between the information used by the SCTs and the computations they perform. Some noteworthy observations enabled by our reformulation are as follows:

1. The global-based and type-based strategies of Ogando *et al.*, as per our reformulation (Section 4.5.2), primarily differ in the interconnection graphs they input. They both use the same computation to create groups, i.e., they compute weak groupings around $\mathcal{Q}$ of the CXG they input. However, in the original paper their computations appear significantly different [27].
2. Livadas and Johnson's type-based strategy is influenced by Ogando *et al.*'s type-based strategy. As per our reformulation (Sections 4.5.3 and 4.5.2), they differ in that, of the respective CXG they input, the former computes strong groupings around $\mathcal{P}$ while the latter computes weak groupings around $\mathcal{Q}$. A difference not explicit in the original work [21].
3. Patel *et al.* and Ogando *et a.* both use interconnection graphs that state which procedure references which types of variables. They both propagate references to a variable of a type to other types in the sub-part/super-part hierarchy. But the two differ in the direction in which references are propagated in the hierarchy. Patel *et al.* propagate the references "upward" while Ogando *et al.* propagate references "downward" (Section 3.1.2).
4. The similarity matrices sim_2, sim_5, and sim_9 computed by Hutchens and Basili (Section 4.1.1), Maarek and Kaiser (Section 4.1.3), and Müller and Uhl (Section 4.3.1) require the same computation, albeit on different WXGs. They all sum the weights on the edges between pairs of nodes of the respective WXGs. The differences in the terminology and symbols used in the original works obscures this similarity.
5. Hutchens and Basili (Section 4.1.1) and Müller and Uhl (Section 4.3.1), both compute the *degree*—the sum of *indegree* and *outdegree*—of nodes in the respective graphs. However, the original works do not refer to it as computation of the degree of a graph.
6. We use the symbols sim and $diss$ whenever a computation yields a similarity or a dissimilarity matrix (as defined in Section 3.3). While the enumeration of this distinction is necessary for SCTs using HAC algorithms [19], we use it for other SCTs as well (see the non-numeric, non-stratified SCTs). In addition to highlighting an important property of these computations, using these symbols also identifies computations that may possibly be used with HAC algorithms.

5.2 Creating new SCTs

SCTs are essentially heuristic. New SCTs may be created by mixing and matching the information and the computations used by different SCTs. The template for a stratified SCT, Figure 3, enables the creation of a parameterized stratified SCT, i.e., a SCT in which the interconnection graph to be used, the formula to compute the

similarity matrix or dissimilarity matrix, and/or the decisions for Steps *s1*, *s2*, and *s3* are parameterized. We have implemented one such parameterized SCT that allows the following selections:

1. The input may be one of two CFGs, either the CFG ψ_{gx} representing the flow of values through global variables, described earlier, or the CFG ψ_{sx}, where $\psi_{sx}(A, x, B)$ is true iff the variable, type, or function x is referred (i.e., used, assigned, or called) by procedures A and B.
2. The computations for Steps *s1*, *s2*, and *s3* may be either those used in single-link HAC or those used in Hutchens and Basili's SCTs [17].
3. If single-link HAC is chosen, it may be applied on one of three matrices sim_2, $diss_1$ or $diss_3$ as described in Section 4.1.1.

This ability to create several SCTs by parameterizing a general template is interesting because SCTs are essentially heuristic. A parameterized SCT provides the ability to experiment with different combinations of computations and to choose the one most suitable for a particular environment. For instance, our parameterized stratified SCT can generate ten subsystem classifications for each software system (two CFGs for each system, two subsystem classifications for each CFG due to Hutchens and Basili's SCT, and three subsystem classifications for each CFG using single-link HAC).

5.3 Evaluating applicability of third-party programs and SCTs

All the SCTs we study, except that of Ong and Tsai [28], can be parameterized on their input interconnection graph, i.e., their computation may be stated in terms of a generic CXG, WXG, or CFG. Such a parameterization makes a SCT independent of any programming language or any environmental context. In addition to creating a class of SCTs from each SCT, this enhances the possibility of using the same SCT for systems written for different programming languages by simply replacing the front end.

The development of a language specific front-end is an expensive task in the implementation of a SCT. One may prefer to use a third-party program analyzer such as Refine [34] or FIELD [35] to perform this function, instead of developing one. These analyzers often differ in capabilities as well as price. The comprehensive list of interconnection graphs presented in Figures 1, 2, and 3 may be used to create guidelines for evaluating whether a third-party tool extracts the information needed to develop one or more SCTs.

Here are a few examples of the information needed for and the applicability of some SCTs. SCTs that use 'type' related information would not be useful with FORTRAN programs since these programs do not have user-defined types. SCTs that use 'global variables' related information may not do well with programs using data abstractions. To develop Ogando *et al.*'s type-based SCT and Patel *et al.*'s SCT one needs the information 'which type is used to define which type' [27, 31,]. A cross-reference information extraction tool that does not provide this information would not be suitable for this purpose. Patel *et al.*'s SCT also requires the count of how many times a relation between two symbols exists, 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.

6 Conclusions

We have presented a framework to describe techniques that decompose the components of a program into subsystems. Such a decomposition, referred to as a subsystem classification, is useful in various contexts during software maintenance. Therefore, techniques for extracting subsystem classifications have been investigated by researchers developing tools to support different maintenance activities, such as program understanding [6], reengineering legacy code [25, 28, 36], propagating changes [23], analyzing error-prone components [37], and identifying objects in source code [21, 27]. Given the differences in the problem domains, these techniques have been presented using different "languages" — terminology and symbols. Our reformulation of these techniques using a consistent "language" is an attempt to dismantle the "Tower of Babel" thus created.

Our investigation of SCTs stems from our interest in developing automated support for identifying modules as a precursor to reengineering legacy code. We feel that these techniques hold promise, even though they are heuristic

and depend primarily on cross-reference information. The use of cross-reference information for organizing the components of a program into modules finds support from the observation that the notion of information hiding is itself defined on the basis of scope and operations on symbols [30], which are essentially cross-reference constraints. It is also supported by Parikh's observation that cross-referencing is one of the most important sources of information for a maintenance programmer [29].

That numerical cluster analysis provides a good heuristic for recovering modules (or objects) may be extrapolated—epigrammatically, not logically—from the very application it was first designed for, namely, to classify animal and plant kingdoms—classification hierarchies commonly used as examples for object-oriented design. Interestingly, numerical cluster analysis has also been used to generate "programs"—sequences of instructions to an architect—from architectural design constraints when designing cities [4], the reverse of what SCTs seek to do.

Acknowledgments

This work has benefitted significantly by the careful reading and comments provided by Anurag Bhatnagar, John M. Gravley, Zhen Huang, Bharat Nedunchalian, Anil K. Vijendran, and the anonymous referees. The work was partially supported by the grant Louisiana BOR LEQSF (1993-95) RD-A-38 and ARO DAH04–94–G-0334. It was conducted in the Software Research Laboratory funded by the grant Louisiana BOR LEQSF (1991-92) ENH-98.

Bibliography

[1] B. L. Achee and Doris L. Carver. A greedy approach to object identification in imperative code. In *Proc. Workshop on Program Comprehension*. IEEE Computer Society Press, 1994.

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

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

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

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

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

[7] T. J. Biggerstaff. Design recovery for maintenance and reuse. *Computer*, pages 36–49, July 1989.

[8] David Carrington, David Duke, Ian Hayes, and Jim Welsh. Deriving modular designs from formal specifications. *Proc. of SIGSOFT'93, Software Engineering Notes*, 18(5):89–98, December 1993.

[9] 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.

[10] Elliot J. Chikofsky and James H. Cross II. Special issue on reverse engineering. *IEEE Software*, 7(1), January 1990.

[11] Song C. Choi and Walt Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, 7(1):66–71, January 1990.

[12] Peter Coad and Edward Yourdon. *Object-oriented analysis*. Yourdon Press, Englewood Cliffs, NJ, second edition, 1991.

[13] 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.

[14] Alfred E. Dunlop and Brian W. Kerninghan. A procedure for placement of standard-cell VLSI circuits. *IEEE Transactions on Computed Aided Design*, CAD–4(1):92–98, January 1985.

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

[16] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1. World Scientific Publishing Company, 1993.

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

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

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

[20] Arun Lakhotia and John M. Gravley. A measure of congruence between subsystem classifications of a software system. In *2nd IEEE Working Conference on Reverse Engineering*. IEEE Computer Society Press, 1995.

[21] Panos E. Livadas and Theodore Johnson. A new approach to finding objects. *Journal of Software Maintenance*, 6(4):in print, September 1994.

[22] 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.

[23] Y. S. Maarek and G. E. Kaiser. Change management for very large software systems. In *Proceedings of Phoenix Conference on Computers and Communications*, pages 280–285, March 1988.

[24] 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.

[25] 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.

[26] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. Technical Report 95-03-02, Department of Computer Science and Engineering, University of Washington, 1995.

[27] 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.

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

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

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

[31] 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.

[32] D. E. Perry. Software interconnection models. In *Proceedings of the 9th International Conference on Software Engineering*, pages 61–69, April 1987.

[33] Dewayne E. Perry and Alexander L. Wolf. Foundations for study of software architectures. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[34] Reasoning Systems, Inc., Palo Alto, CA. *Refine User's Guide*, 1992.

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

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

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

[38] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architectures and tools to support them. Draft, 1994.

[39] Ignacio Silva-Lepe. An empirical method for identifying objects and their responsibilities in a procedural program. In *Technology of Object-Oriented Languages and Systems Europe Conference*, pages 136–149, Versailles, France, 1993. Prentice-Hall. Also available as technical report NU-CCS-93-2, Northeastern University.

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

[41] Richard C. Waters and Elliot J. Chikofsky. Special issue on reverse engineering. *Communications of the ACM*, 37(5), May 1994.