Cross References are Features

Robert W. Schwanke and Michael A. Platoff
Siemens Corporate Research, Inc.
755 College Rd. East
Princeton, NJ 08540
USA

1. Introduction

When a software system is developed by a large team of programmers, and has matured[1] for several years, changes to the code may introduce unexpected interactions between diverse parts of the system. This occurs because the system has become too large for one person to fully understand, and the original design documentation has become obsolete as the system has evolved. Symptoms of structural problems include too many unnecessary recompilations, unintended cyclic dependency chains, and some types of difficulties with understanding, modifying, and testing the system. Most structural problems cannot be solved by making a few "small" changes, and most require the programmer to understand the overall pattern of interactions in order to solve the problem.

The ARCH project at Siemens Research is building an "architect's assistant" for a software maintenance environment. ARCH will help the software architect analyze the structure of an existing system, specify an architecture for it, and determine whether the actual software is consistent with the specification. Since the system's structural architecture may never have been formally specified, we want ARCH to be able to "discover" the architecture by automatically analyzing the existing code. It should also be able to critique an architecture by comparing it to the existing code and suggesting changes that would produce a more modular specification.

A common approach to structural problems is to treat cross-reference information as a graph, in which software units appear as nodes and cross-references appear as edges. Then various methods, both manual and automatic, may be used to analyze the graph. Recent work by Maarek and Kaiser [1] and Selby and Basili [2] has used *clustering methods* to summarize cross-reference graphs, by clustering nodes into groups, and then analyzing edges between groups.

The ARCH project is developing a new, complementary set of analysis methods, based on the idea that cross references be represented as *features* of the objects they connect. Doing so allows us to use similarity measures[2] based on shared features. This in turn allows us to use *conceptual clustering* methods, originally developed for classification, pattern recognition, information retrieval, and machine learning, and apply them to software analysis.

We have built a detailed cross-reference extractor and a conceptual clustering tool, and are using them to analyze cross-reference graphs for several kinds of software maintenance problems. Our preliminary results suggest that these methods can help reduce unnecessary recompilations, summarize complex structure graphs, and improve modularity. We plan to develop interactive techniques that combine the bookkeeping abilities of the computer with the deep knowledge of the maintainer, to produce even better solutions.

This paper discusses some structural problems that occur frequently in mature systems, describes our feature representation for cross-references, presents the prototype *conceptual clustering* algorithm we are

[1] The word "matured" reflects our belief that old software is worth maintaining because of stability, refinement, and customer loyalty, which only come with experience.

[2] The literature of classification normally uses *dis*similarity measures. We find this term awkward, and will use it only where necessary, relying on the intuitive correspondence between similarity and dissimilarity.

using, and describes how the technology can be used to attack the structural problems. We include preliminary results of our experiments, and details of some planned experiments.

2. Structural Problems in Large Systems

This section discusses three examples of structural problems that occur frequently in large, mature software systems.

2.1. Structure Visualization

Many maintenance tasks require the programmer to contend with graph-like information, such as cross-references, data flow, compilation dependencies, and call graphs. Many programmers like to draw these graphs and use spatial relationships and edge densities to help them understand global characteristics of the information. However, when the graph becomes too large, the global structure becomes lost in the details, even when good heuristic layout algorithms are used.

To visualize a large graph, the programmer must group the nodes of the graph into clusters, where he can think of each cluster as a single conceptual "chunk" of the code, and then draw edges only between chunks.

In section 6 we will describe an experiment in which we used a *conceptual clustering* algorithm to automatically collect "chunks" of a large call graph, and help the analyst label them with meaningful names. The result appears to be an effective aid to graph understanding.

2.2. Compilation Dependencies

In large systems, controlling the compilation-time dependencies between files can have a significant impact on many aspects of maintenance. Adams *et al* have recently analyzed change logs for a carefully-engineered, new system, written in Ada, and concluded that more than half of compilations were unnecessary [3]. This situation can easily become aggravated if programmers do not take adequate care when grouping utility code (such as macros and type definitions) into files. Many projects have a "catch-all" file of widely-used declarations. Maintainers are unwilling to create a new file to contain a new declaration, because of the nuisance of changing *makefiles*, notifying the configuration management team, and so on. Also, they have no good way of knowing when it is time to start a new file. Instead, they place each new declaration in an existing file. Consequently, the catch-all file may change frequently, causing widespread recompilation. Meanwhile, each module is actually using a smaller and smaller fraction of the declarations appearing in it. Although Tichy's *smart recompilation* [4] would alleviate some of the recompilation cost, the conglomeration of loosely-related declarations would continue to cloud the system structure.

Compilation dependencies are also very important during system integration, because in real-life projects, inconsistency between files is the rule rather than the exception. Integration planners need to be sure that the files they are integrating interact only in limited ways, so that they can get at least part of the system working. Each dependency path between files represents an opportunity for a syntactic interface error, which has the potential to disrupt a system integration step.

The structural problem we will explore in section 7 is the problem of dividing a "catch-all" include file into a "reasonable" set of smaller files, in a way that substantially reduces the amount of code each module must include.

2.3. Modularity

Modularity is generally believed to have a significant impact on testability, maintainability, and understandability. As difficult as this belief is to test (e.g. [5]), modularity is a well-established qualitative goal of software design.

According to Parnas's information-hiding principle [6], a good modular decomposition is one in which each module encapsulates design decisions that are likely to change later, typically by implementing the decisions in a set of declarations that are hidden in a private scope.

Unfortunately, unless a system's module structure is specified separately from the code, and *enforced*, it tends to deteriorate over time. An individual programmer may add a dependency between two modules, to solve a particular problem, without causing substantial difficulties. However, dozens of additions over several years eventually make the system excessively hard to modify, test, or understand. Sometimes, there is an attempt to reorganize the system to improve its modularity, but since understanding is so hard, reorganizing is even harder.

For example, consider the Siemens BS2000 Operating System. It comprises 1 million lines of code, has been maintained for 15 years, and currently employs 300 programmers. A few years ago the project management decided to partition the system into modules, in a way that would let them sell compact, customized configurations. In order to do the partitioning, they decided that they first needed an ac-

curate specification of the current structure. The painstaking process of creating this specification took two full years. The ongoing project includes a graph editor for writing and displaying the specification, and a validation tool for detecting architecture violations in the code. These tools will help them keep the architecture and the code consistent, by immediately reporting problems, so that they can be addressed before they get out of hand.

In section 8 we will discuss how to analyze the interconnection structure of a system, to identify problems and suggest improvements.

3. Connections vs. Shared Neighbors

Previous attempts to analyze program structure by clustering have used similarity measures based on strength of connection. They represent software objects as vertices in a graph, and connections between objects as edges in the graph. Then they define the "strength" of connection between two objects as the number of edges connecting them.

Maarek and Kaiser use connection strength clustering for integration planning. They propose to create an integration plan by clustering the software units into larger and larger clusters, forming a cluster tree. Each node of the tree would be an integration step, in which the clustered objects were tested together, resolving all inconsistencies among them. Maarek and Kaiser define a connection as any identifier that is defined in one unit (or cluster) and used in another. Then they define a similarity measure between two clusters based on connection strength between them.

Selby and Basili use connection strength clustering to identify error-prone code. They define a connection between two units as any variable that passes data from one to the other. Then they define the "goodness" of a cluster as the ratio of its "cohesion" (number of connections within the cluster) to its "coupling" (number of connections between the cluster and other objects).

Although these projects have successfully applied their similarity measures to software maintenance problems, connection strength does not adequately capture *design similarity* between software units.

Consider, for example, the *Sine* and *Cosine* routines from a mathematical software library. One would expect that whatever implementation tricks were used to make one of them efficient should also be used in the other, yet one would be surprised if either one of them actually called, or passed data to, the other. On the other hand, we would expect that many of the other software modules that called the *sine* routine would also call the *cosine* routine, and vice versa. This situation is portrayed in a hypothetical call graph shown in figure 3-1. A similarity

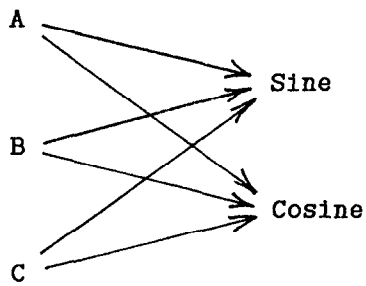

Figure 3-1: Call Graph With Parallel Structure

measure based on connection strength would determine that *Sine* is more similar to *A*, *B*, and *C* than it is to *Cosine*. Clustering the two most similar nodes might produce the graph in figure 3-2, modulo permutations of {A, B, C} and {Sine, Cosine}. Clearly,

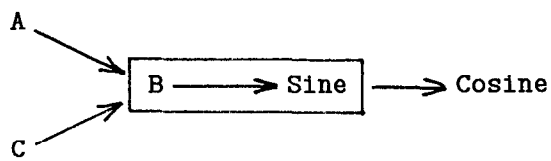

Figure 3-2: Summarizing By Connection Strength

this is unacceptable! We need, instead, a similarity measure that recognizes the parallel structure apparent in the figure. Measures based on *shared neighbors* do this very well. In figure 3-1, both *Sine* and *Cosine* have the neighbors *A*, *B*, and *C*. Conversely, *A*, *B*, and *C* all have the neighbors *Sine* and *Cosine*. Clustering the two nodes that share the most neigbors would produce the graph in figure 3-3

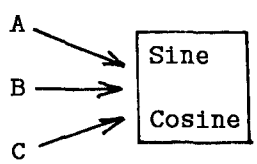

Figure 3-3: Summarizing By Shared Neighbors

4. Cross References as Features

Our analysis methods represent edges in a graph as *features* of the nodes they connect, and measures similarity of nodes by looking at which features two nodes have in common, and which are different. We justify this measure of similarity by looking at its im-

plications for structure problems in software engineering.

4.1. Representation

Consider a graph consisting of a set of nodes {*A, B, C, ...*} and a set of edges {*(X, Y)*}. We represent each node as an object with the same name, and represent each edge *(X,Y)* by giving the object *X* a feature *#Y*, and giving the object *Y* a feature *&X*. We use two different kinds of feature names, *&M* and *#M*, to distinguish the names of an object's predecessors and successors in the graph, respectively.

In software engineering, these graphs represent cross reference information. The set of features {*#M*} of object *X* represent the non-local names occurring in it (its *names-used*), and the set of features {*&M*} represent the names of other software units that use the name *X* (its *user-names*). In our previous example, the *Sine* and *Cosine* routines both have features {*&A, &B, &C*}, and the routines *A, B,* and *C*, have features {*#Sine, #Cosine*}. The ARCH project uses fully scoped names, so that all software units have names that are unique system-wide.

4.2. Similarity Measures

By representing cross-references as features, we can easily identify neighbors that are common to (shared by) two nodes, by comparing their feature lists. We can define several interesting similarity measures, by counting the number of shared neighbors, non-shared neighbors, or both. We can also define aggregate measures, for measuring similarity between two groups of objects, by looking at the frequency with which features occur in the groups. We are currently studying similarity measures derived from information retrieval, from information theory, and from machine learning research.

Shared-neighbor similarity measures do not replace or subsume connection strength measures. If two nodes are connected, but have no shared neighbors, the similarity between them will be zero. However, for some experiments, it makes sense to treat the definition site of an identifier as also being a use-site for that identifier. In those situations, we give each object *X* the feature *#X*, and sometimes also *&X*. This gives it some similarity to its neighbors. Future research should explore composite similarity measures based on both connection strength and shared neighbors.

4.3. Software Engineering Rationale

There are two kinds of reasons to cluster software based on shared neighbors: structural reasons and semantic reasons.

The structural reasons arise in situations where the patterns of interaction are intrinisically important. For example, when partitioning an include file into smaller files, declarations that are used in all the same modules should be placed in the same partition. Similarly, when studying a call graph, forming a group out of procedures with the same callers and callees allows one to simplify the graph without suppressing much information.

The semantic reasons arise because the neighbors of a software object tell you what it's built out of, and how it's used. For example, if you noticed that a certain module called the procedures `FileOpen`, `FileRead`, and `StringCompare`, you might guess that it was doing some kind of lexical analysis on the contents of files.

Even if you don't know exactly what information a name represents, seeing the same name occurring in two software units suggests that their implementations are related. This may be due to a shared variable, macro, type, or procedure; in each case, it means that they both rely on the functional specification of the shared name.

This shows the relationship between shared neighbors and the Parnas information-hiding principle: if a group of software units share a set of data types, variables, macros, and/or procedures, which few other units use, the group should be considered as a potential module.

Most of the rationale given above can be reworded to support the hypothesis that a procedure or other software unit can also be characterized by where it is used (its *user-names*). For example, if most of the procedures that invoke the macro `XtWindow` also invoke `XtDisplay`, you might guess that the two macros are related -- as they are, in MIT's X Toolkit [7].

We conclude from the structural and semantic arguments above that clustering based on shared neighbors is likely to be a useful way to analyze structural problems in large software systems.

5. The Clustering Procedure

In this section, we describe the clustering procedure we are using in our current experiments. The procedure is an example of a hierarchical, ascending classification method, specialized to be a *conceptual clustering* method. It produces a tree of classes and subclasses of the objects. Each class has a description, consisting of the size of the cluster and, for each possible feature, the proportion of cluster members that have the feature. The algorithm presented is a prototype; rather than try to give it a mathematical

justification, we will discuss the issues we are currently exploring as we redesign it.

5.1. Hierarchical Ascending Classification

In general, a hierarchical ascending classification (HAC) algorithm forms clusters (classes) of objects in a bottom-up fashion, by first forming small clusters of closely-related (or very similar) objects, then combining the small clusters into larger clusters, finally forming a classification tree, whose leaves are the original objects, and whose interior nodes are the classes. (We use the terms "cluster", "class", and "category" somewhat interchangeably.) The interested reader should consult Maarek's fine overview of HAC methods [8].

HAC algorithms may be contrasted with partitioning algorithms, which divide a set of objects into two or more classes, then recursively divide each class. At each level, a partitioning is sought that maximizes the similarity of objects within each class, and maximizes the differences between classes.

Both HAC and partitioning methods optimize early decisions at the expense of later decisions. The partitioning methods may form excellent top-level classes, but the choice of top-level partition may prevent the formation of the "best" sub-classes deeper in the tree. Conversely, HAC methods form small clusters first, and thereby constrain the possible large clusters.

For our applications, we have concluded that the low-level clusters are more important, and we are therefore using HAC methods. For example, a well-modularized system should have sharply-defined first-level modules, even if the top-level subsystems are a little less "pure" because of it.

5.2. The Algorithm

We present the control structure of the algorithm first, then explain the key computations in more detail.

The Arch Batch Clustering Algorithm

Purpose: form a classification tree T over the set of nodes $\{N_b\}$. Each subtree represents a category, containing the nodes named at its leaves.

1. For each N_i, create tree T_i consisting of the single leaf N_i, and place it in the candidate set C.
2. **Repeat**
   a. Find the most similar pair of trees in C, say T_x and T_y, and remove them from C.
   b. Create tree T_z with two children, T_x and T_y, and add the new tree to C.

   Until C contains only one tree, say T_{root}.
3. *Massage* (T_{root})

Massage is a recursive procedure to increase the average branching factor of a tree by eliminating low-utility interior nodes. Utility is measured by a category utility function *CU* (*T*). Eliminating an interior node entails promoting its children to be children of its parent.

Massage (*T*):

1. **Loop**
   a. Find a child T_c of T such that *CU* (T_c) is minimal.
   b. **If** eliminating T_c would increase the average category utility of the children of T, **then**
      eliminate T_c
      else exit loop

   end loop
2. For *every* child T_e of T,
   Massage (T_e)

The category utility function *CU* is the product of the size and "purity" of a category. "Purity" is the sum of squares of the feature frequencies, i.e. of the probability that a member of the category has a given feature. It favors categories in which most members share many features, and it favors large categories.

The two most similar categories are actually selected by finding the two least *dis*similar ones. The dissimilarity of two categories, X and Y, is a function of their category utilities:

$$DisSim\ (X,Y) = CU(X) + CU(Y) - CU(X \cup Y)$$

Two categories with identical feature frequencies will have no dissimilarity between them.

The category utility function was adapted from Cobweb's CU function, on the assumption that only "present" features were significant, and not "absent" features.

5.3. Computational complexity

Our naive implementation of the algorithm has cost $O(n^3 f)$, where n is the number of nodes, and f is the average number of features per node. This is due to the n^2 tree-to-tree comparisons required to find the two most similar trees. A more efficient algorithm

might be obtained by using inverted indices to analyze only the nodes that actually share at least one feature with a given node, and by using feature frequency information to optimize the order of comparisons.

Fortunately, the algorithm's complexity has not yet overwhelmed our computing resources. We have been analyzing 100-node graphs in elapsed times of under 10 minutes, on a Sun 3 workstation.

5.4. Conceptual Clustering

A conceptual clustering method is a clustering or partitioning method that produces not only the clusters themselves, but also *descriptions* or *explanations* of the clusters. For example, it might produce clusters of insect specimens, and label one of them "have six legs and four wings". The cluster is said to represent a *concept* both by *extension* (listing examples) and *intention* (the description).

The Cobweb system [9] is a good example of a conceptual clustering system; it gave us many of the basic ideas for our current work. In Cobweb, the description is simply a list of how many times each feature occurs among members of the cluster. ARCH uses the same type of description. Since the clusters are formed around shared features, the description accurately explains why the cluster was formed. We have found that these descriptions help us to attach useful names to the "concepts", as will be discussed in section 6. If ARCH had measured similarity by connection strength instead, then the "explanation" would be a list of connections. We rejected this approach because we found that lists of connections did not give us useful insights into a system's structure.

Many conceptual clustering methods include submethods that select only those clusters that have simple and/or useful descriptions. If the cluster tree is supporting some knowledge-based application program, for example, the choice of clusters may be oriented toward making the application as efficient as possible. The *Massage* procedure performs this function in ARCH, selecting clusters to optimize average category utility. We do this to obtain "reasonable-and natural-sized" clusters. For our modularity experiments, we are exploring acceptance predicates that prefer clusters with good information-hiding qualities, as described in section 8.

5.5. Potential Improvements

We are currently exploring several ways to improve the algorithm:

- **Feature weighting by frequency**: the current algorithm forms clusters around frequently-occurring features more than around rare ones. The opposite would be preferable, because, for example, the "secrets" of small modules must necessarily be rare. We think that systematically weighting features in proportion to their rarity will produce better results.
- **Similarity measures and category utility**: There are many valid ways to define similarity between groups of objects. We are exploring several of them. We are also looking at different definitions of "purity".
- **Category elimination**: our current method is based on optimizing average category utility. We are also looking at application-specific heuristic predicates, which analyze a subtree node in context and accept or reject it.
- **Massaging sequence**: our current implementation massages the tree top down. This produces a "boundary effect" at the bottom of the tree, where the average branching factor is much less than in the rest of the tree. We expect that bottom up or globally-controlled sequencing will produce better trees for the applications we are studying. The bottom-up algorithm is the same as the *Massage* algorithm given, except that step 2 is performed before step 1. A globally-controlled algorithm would evaluate all nodes in the classification tree, and repeatedly eliminate the least desirable one until no further benefit is obtained.
- **Adaptation**: Allowing human architects to bias the weights of features, by providing feedback on classification decisions, should lead to improved classification.

6. Summarizing a Call Graph

Clustering by connection strength is attractive for summarizing call graphs, because it promises to find subgraphs containing large numbers of internal connections, with relatively few connections between subgraphs. However, because of the partial-ordering characteristic of call graphs, this approach would tend to find "vertical" groups first, because most connections would occur between nodes in different levels of the graph. Sometimes that kind of analysis is useful, but we believe that, for understanding the overall structure of a call graph, it is more important to first find "horizontal" groups of procedures, representing layered abstractions in the system, even if members of a group do not directly call one another.

A preliminary clustering experiment supports this hypothesis. The experiment consisted of forming a

cluster tree, labelling the interior nodes, and then displaying the graph in various summarized forms using the Edge graph browser [10].

The call graph came from the TML subsystem of the DOSE structure editor generation system [11]. TML is a recursive descent program interpreter, with associated interactive debugging commands. The subsystem contains 82 procedures. Its internal call graph contains 155 edges. A legible diagram of the call graph measures 8" by 30". It is complicated enough that its "overall structure" is not obvious, although a knowledgeable maintainer could trace individual paths through it without trouble.

For this experiment we treated each procedure as using its own name, as well as defining it, so that there would be some similarity between a procedure and its callers. We represented this, in the manner discussed earlier, by giving each procedure X the feature $\#X$.

The clustering algorithm was presented with 392 features (2 per edge, plus a self-reference for each node). It created a subsystem tree consisting of 29 clusters, including the root cluster comprising the entire system, yielding an average branching factor of 3.8. The run time was 323 seconds on a 12 Mbyte diskless Sun 3 workstation.

The labelling step in this experiment was performed manually, but with substantial machine help. The machine produced a feature summary for each cluster, listing how many times each feature occurred in the cluster. By reading these lists, and drawing on our knowledge of the code, we easily recognized the common design properties of the clustered procedures, and wrote short descriptive titles for 26 of the 29 clusters.

To demonstrate that the clustering was useful for understanding the graph, we fed the cluster data and the original call graph to Edge for display. From the data we generated summaries of the graph at several different levels of detail, forming a tutorial sequence presenting details of the graph in small, manageable increments. Figure 6-1 shows a high-level summary, dividing the graph primarily into *CoarseControl* and *ExpressionUsers*, with two pivotal routines connecting them. Figure 6-2 shows more detail, showing that expression evaluation is isolated from the rest of the system. Subsequent frames of the tutorial sequence show more and more detail, corresponding quite well to our own knowledge of the code.

We conclude from this experiment that clustering objects in a software interconnection graph according to their *names-used* and *user-names* is useful for understanding the overall structure of the graph.

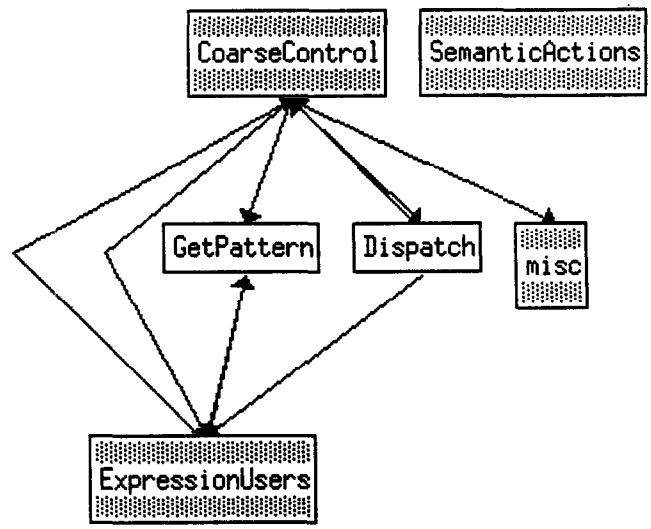

Figure 6-1: Highly Summarized TML Call Graph

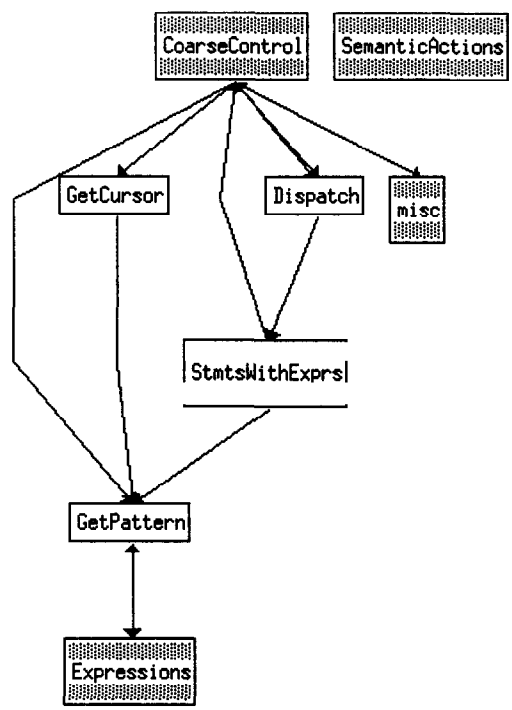

Figure 6-2: More Detail of TML Call Graph

The astute reader might challenge the validity of our labelling method, noting that we already understood the code. We agree that we do not yet have a labelling method for unfamiliar code based entirely on features. However, the experiment does show that our clustering algorithm can discover human-useful concepts, even though it cannot give them names. It also

shows that providing the feature summary makes the labelling process much easier than labelling without knowing the features. We plan to explore automatic, heuristic labelling methods that identify the distinguishing features of a category and construct a label out of them.

7. Splitting an Include File

To split up an *includes* file that has become too large, one must satisfy two goals: keep together those declarations that are conceptually related, and separate those declarations that are used in different files.

These two needs can be achieved by performing the classification in two stages. First, form small clusters of units that are closely conceptually related, based on all of their names-used and user-names. Have a human approve all clusters that are kept. Then, group the small clusters into larger clusters based only on user-names; specifically, the names of the *files* that use each cluster.

The second stage requires that rare features outweigh common features in determining clusters, because rare features represent files that use few clusters, and therefore have much to gain by having all those clusters in the same partition. In contrast, common features represents files that use many different clusters, and therefore have little to gain by splitting the include file.

A special feature of this application problem is the need to decide exactly how many top-level categories are needed in the tree. This decision could be based solely on minimizing recompilation cost, or on some combination of recompilation cost and the cost of administering large numbers of files. Recompilation cost is a function of how much total text must be processed, and how many times files must be opened. Total text decreases as the number of categories increases, because each declaration is included fewer times. However, more categories implies that more *FileOpen* operations will be required. The evaluation function could be used automatically, replacing the average category utility computation in the *Massage* algorithm. Or, it could be provided to the human architect as advice.

8. Improving Modularity

ARCH defines the "architecture" of a system to be a subsystem tree plus a list of the allowed export-import connections among subsystems. Unlike some systems, ARCH allows constraints to be specified between any pair of subsystems, not just between siblings at the same level in the subsystem tree.

Although ARCH can use clustering to propose a completely new architecture for an existing system, this will not be its primary service. It is rare that a software project has lost *all* of its architecture information. Usually, the system is at least divided into files, and the files are compiled and linked in some non-uniform way that hints at an underlying architecture. It is also unlikely that an ongoing project would accept a completely machine-generated architecture, because there are usually non-technical reasons for some architectural decisions, and because abruptly adopting an unfamiliar architecture would cause an unacceptable disruption in the project.

Therefore, ARCH will take advantage of whatever architecture specifications are available. It will evaluate the "goodness" of the existing architecture specifications, with respect to the actual code, and propose changes to parts of the architecture while accepting other parts as given. For example, it will identify "misplaced" procedures that should be moved to a different subsystem, and identify potential new modules to be formed from parts of existing modules.

We also hope that, through interactions with human architects, ARCH can "learn" which features are most important to the architecture, and by using this information to weight the features, provide analyses that are more useful to the architects.

Both evaluating existing architectures and proposing new modules require that we first validate our similarity measures, by showing that they measure desirable properties of real programs. The next three sections describe the validation effort, and then outline how ARCH will use similarity and clustering to assist human architects.

8.1. Validating the Similarity Measure

We are validating the similarity measures by showing that similar procedures in well-modularized systems are likely to belong to the same subsystem.

We use a *nearest neighbors* test, based on Ellen Voorhees's test for the cluster hypothesis in information retrieval [12]. (The cluster hypothesis states that documents should be clustered because similar documents tend to be relevant to the same queries. Substitute "procedures" for "documents", "belong" for "be relevant" and "subsystem" for "query".) A software unit's *nearest neighbors* are the other software units to which it is most similar. A unit is well-placed if most of its nearest neighbors are in the same module, and mis-placed if most of its nearest neighbors are in other modules. We call mis-placed units *mavericks*. The *k nearest neighbors test*, for our problem domain, is this: for each software unit U

and each of its k nearest neighbors $NN(U,i)$, $i = 1..k$, count how often Module(U) is the same as Module(NN(U,i)).

We are using the nearest neighbor test to analyze two real software systems for which the subsystem structure is already known. The first, DOSE [11], is a mature, inadequately-structured system. The second, the Tiled Window Manager (TWM) [13], is a young, carefully structured system. Each contains many thousands of lines of code, and on the order of 1,000 procedures. In both cases we are using "source file" to approximate "module" for the purposes of the test, on the assumption that good C programmers normally use files to define the modularity of their systems. We will soon begin evaluating a third system, the BS2000, for which an architecture has been carefully specified after 15 years of maintenance. This will be the most important test, because the architecture represents human performance of exactly the task we want ARCH to do.

Although our experiments are incomplete, we are finding that a procedure and its nearest neighbor are located in the same file at least 70% of the time. More interestingly, we are finding that, when the nearest neighbor is in a different file, the reasons are often enlightening. One common reason is that the procedure body is very small (sometimes even empty!), making it not very similar to *any* other function. Another reason is that two functions perform identical services in different contexts. Yet another is that an over-large module has been split across two files without regard for information hiding. Occasionally we find ill-conceived functions that should be divided into several subroutines placed in different modules, and occasionally we find global variables and data structures in need of better modularity.

From these experiments we are gaining confidence that the similarity measure correlates strongly with the modularity of carefully-designed code, and that when the nearest neighbor test identifies a maverick, studying the maverick is likely to reveal important architectural characteristics and/or problems.

8.2. Suggesting Incremental Improvements

When validation is complete, ARCH will incorporate the similarity measures into an architecture *critic* facility, which will analyze existing system architectures, pointing out anomalies and suggesting improvements.

The simplest type of improvement is a single-unit move. ARCH will repeatedly identify a maverick, and suggest moving it into the module of one of its nearest neighbors. Naturally, only a human architect can determine with certainty whether to move a unit. Should the architect disagree with a move, ARCH will adjust the weights of features, as described before, to improve the similarity between the maverick and the other members of its module. ARCH will also record the architect's decision so that it does not make the same suggestion again later.

A second kind of improvement entails deleting subsystem nodes, much like is done in the *Massage* procedure. ARCH will incorporate heuristic procedures to review the features of a subsystem, its children, its parent, and its siblings, and assess how much information hiding it is actually doing. It will suggest eliminating subsystem nodes that hide little.

A third kind of improvement entails adding subsystems. ARCH will propose additions by examining each node in the subsystem tree and attempting to cluster its children. Cluster selection criteria, as are used in *Massage*, will determine whether new nodes would improve the structure.

8.3. Large Scale Restructuring

Making incremental changes to an existing architecture can lead only to locally optimum architectures. To attain more global optima, larger portions of a system may need radical restructuring.

ARCH's clustering algorithm, applied to full cross-reference data for a set of software units, produces a classification tree that can be used as a subsystem tree. By weighting features in proportion to their rarity, it will tend to concentrate rare features in small subtrees, leaving common features more spread out. Thus, the algorithm will tend to minimize the scopes of infrequently-occurring identifiers. Although one cannot always confine a rarely-used identifier to a single, small subsystem, one can usually restrict it to be used only in a small number of small subsystems. On the other hand, a commonly used identifier cannot possibly be confined to a small subsystem, because it is used by too many other software units.

Despite these pleasing prospects, ARCH will not produce architectures completely automatically. Instead, ARCH will present each of its proposed subsystems to the architect, as it is formed. If the architect approves, the algorithm proceeds in the same fashion as if it were fully automatic. If the architect disagrees, a dialog will uncover the features that characterize the disagreement. The weights of those features will be tuned to fit the judgements that have been made so far, and then the classification will continue. We hope that a small number of "no" answers by the human will permit enough tuning to substantially reduce the total number of "no" answers needed.

ARCH will also provide some help with naming the new subsystems, based on previous subsystem structure. ARCH will compare each new subsystem to each subsystem from the "old" architecture, looking at both the software units they contain and the features that describe them. It will then suggest naming the new subsystem after the most similar "old" subsystem.

9. Conclusions

Representing cross references as features allows us to apply feature-based similarity measures and conceptual clustering techniques to large software systems. The measures are complementary to connection-strength measures, but do not necessarily subsume them. The features allow automatic construction of useful category descriptions. Shared-neighbor similarity appears to be correlated to the way that C programmers prefer to partition their code into files, although further experiments are needed. The mavericks ARCH finds in DOSE and TWM point to interesting properties and problems in those two systems. The next generation of ARCH will be an interactive architecture editor and critic, using both maverick analysis and clustering methods to help human architects improved their designs.

10. Acknowledgements

Gail Kaiser encouraged us to pursue this project before we were sure we should. Francie Newbery kindly supplied us with the source code for Edge, and lots of cooperation as Rita Altucher stress-tested and debugged it with large graphs. George Drastal introduced us to conceptual clustering and told us about Cobweb. Chris Buckley and Vivek Gore have been implementing and running the nearest-neighbor tests.

1. Yoelle S. Maarek and Gail E. Kaiser, "Change Management in Very Large Software Systems", *Phoenix Conference on Computer Systems and Communications*, IEEE, March 1988, pp. 280-285.

2. Richard W. Selby and Victor R. Basili, "Error Localization During Software Maintenance: Generating Hierarchical System Descriptions from the Source Code Alone", *Conference on Software Maintenance -- 1988*, IEEE, Oct 1988.

3. Rolf Adams, Annette Weinert and Walter Tichy, "Software Change Dynamics *or* Half of all Ada Compilations are Redundant", *European Software Engineering Conference*, 1989.

4. Walter F. Tichy, "Smart Recompilation", *ACM Trans. Programming Lang. and Systems*, Vol. 8, No. 3, July 1986, pp. 273-291.

5. Virginia R. Gibson and James A. Senn, "System Structure and Software Maintenance Performance", *Communications of the ACM*, Vol. 32, No. 3, March 1989, pp. 347-358.

6. David L. Parnas, "On the Criteria To Be Used In Decomposing Systems Into Modules", Tech. report, Computer Science Department, Carnegie-Mellon University, August 1971.

7. Joel McCormack, Paul Asente, and Ralph R. Swick, "X Toolkit Intrinsics -- C Language X Interface", Tech. report, Massachusetts Institute of Technology, 1988.

8. Yoelle S. Maarek, *Using Structural Information for Managing Very Large Software Systems,* PhD dissertation, Technion -- Israel Institute of Technology, January 1989.

9. Douglas Fisher, "Knowledge Acquisition Via Incremental Conceptual Clustering", *Machine Learning*, Vol. 2, No. 2, 1987, pp. 139-179.

10. Frances J. Newbery, "EDGE: An Extendible Directed Graph Editor", Tech. report 8/88, Universitaet Karlsruhe, 1988.

11. Peter H. Feiler, Fahimeh Jalili, and Johann H. Schlichter, "An Interactive Prototyping Environment for Language Design", *Proceedings of the Hawaii Conference on System Sciences*, January 1986.

12. Ellen M. Voorhees, "The Cluster Hypothesis Revisited", *Research and Development in Information Retrieval*, ACM SIGIR, June 1985.

13. Ellis S. Cohen, Edward T. Smith, and Lee A. Iverson, "Constraint-Based Tiled Windows", *First International Conference on Computer Workstations*, IEEE Computer Society, November 1985.