CRAFT: A Framework for Evaluating Software Clustering Results in the Absence of Benchmark Decompositions

Brian S. Mitchell and Spiros Mancoridis
Department of Mathematics & Computer Science
Drexel University, Philadelphia, PA, USA
{bmitchel, smancori}@mcs.drexel.edu

Abstract

Software clustering algorithms are used to create high-level views of a system's structure using source code-level artifacts. Software clustering is an active area of research that has produced many clustering algorithms. However, we have seen very little work that investigates how the results of these algorithms can be evaluated objectively in the absence of a benchmark decomposition, or without the active participation of the original designers of the system.

Ideally, for a given system, an agreed upon reference (benchmark) decomposition of the system's structure would exist, allowing the results of various clustering algorithms to be compared against it. Since such benchmarks seldom exist, we seek alternative methods to gain confidence in the quality of results produced by software clustering algorithms.

In this paper we present a tool that supports the evaluation of software clustering results in the absence of a benchmark decomposition.

1. Introduction & Background

Maintaining large and complex software systems is a daunting task. Poor software maintenance practices often lead to a system's premature retirement, or necessitates significant restructuring in order to keep the source code in manageable condition. There are many causes for these poor maintenance practices. Little can be done about practical issues such as limited maintenance budgets, lack of access to the original developers and designers of the system, and short development schedules. However, if software maintainers have a thorough understanding of the underlying system structure, it is more likely that changes to the source code can be performed without deviating from the system design and architecture. Additionally, if the source code structure is documented, new maintenance developers will require less time to become productive.

The best way to gain insight into a system's structure is to have access to accurate documentation. Unfortunately system documentation is often not up to date, if it exists at all. One way that researchers have addressed this problem is by developing software clustering algorithms and tools that produce high-level views of a system's structure directly from its source code. Because clustering techniques use a variety of criteria to decompose a software system into clusters, it is not surprising that these algorithms produce different results when applied to the same system.

The above issues further complicate things for software practitioners who are trying to understand the structure of complex systems. A manual analysis of the source code is tedious, and may not even be possible, because of its large number of components and dependencies. Clustering algorithms automate this analysis.

Many of the clustering techniques published in the literature can be categorized by the way they establish clusters. Hutchens and Basili [7] developed an algorithm that clusters procedures into modules by measuring the interaction between pairs of procedures. Schwanke et al. [16, 17] introduced the notion of using design principles such as low coupling and high cohesion to create clusters. Choi and Scacchi [4] describe a clustering technique based on maximizing the cohesiveness of clusters by evaluating the exchange of resources between modules. Hausi Müller et al. [15] implemented several software clustering heuristics in the Rigi tool that (a) measure the relative strength between interfaces, (b) identify omnipresent modules, and (c) use similarity of module names. Clustering based on similar patterns in implementation information (*e.g.*, module file names) has been investigated by Anquetil et al. [2, 3]. Concept analysis [10, 19, 1] has also been explored in the software clustering research. Our research is based on using optimization techniques [12, 5, 11, 14] to determine clusters.

Now that a plethora of clustering approaches exist, the validation of clustering results is starting to attract interest from the Software Engineering research community. Many of the clustering techniques published in the literature present case studies where the results are evaluated by the authors or by the developers of the system being studied. This evaluation technique is very subjective. Moreover, the literature often does not describe the types of systems for which a clustering technique does not perform well. For example, our clustering technique, named Bunch [12, 5, 11, 14], forms clusters based on maximizing the cohesiveness of the clusters, while minimizing the inter-cluster coupling. While we think that this technique is good for many types of systems, it is not suitable to all systems. As an example, Bunch may not provide good results for systems whose architecture is event driven (*e.g.*, UI code), or whose architecture loads system resources dynamically[1].

The issues described above should be considered, especially when the decompositions produced by different clustering algorithms differ dramatically. Recently, researchers have begun developing infrastructure to evaluate clustering techniques by proposing similarity measurements [2, 3, 13]. These measurements enable the results of clustering algorithms to be compared to each other, and preferably to be compared to an agreed upon "benchmark" standard. A high similarity value should provide confidence that the clustering algorithm is producing good decompositions.

The remainder of this paper is dedicated to techniques for evaluating the results of clustering algorithms when no standard benchmark exists. We present a tool we developed that provides a framework for evaluating clustering techniques. We hope that our tool will be of value to researchers who are creating or evaluating clustering algorithms.

2. Establishing Confidence in Software Clustering Results

In this section we address the problem of how to gain confidence in a software clustering result when a reference benchmark standard is not available for comparison. Although significant research emphasis has been placed on clustering, we have seen little work on measuring the effectiveness of these techniques. The following list outlines the current state of practice for evaluating software clustering results:

- Software clustering research often revisits the same set of test cases for evaluation (*e.g.*, `linux`, `mosaic`, etc.). This is appropriate because the structure of these systems is well understood. However, for industry adoption of software clustering technology, researchers must show that these tools can be applied to a variety of systems successfully.

- There are many clustering tools and techniques. There is value to this diversity, as certain clustering techniques produce better results for some types of systems than others. However, no work has been done to help guide end-users to those techniques that work best for their type of system.

- The interpretation of software clustering results tends to focus on the overall result (*e.g.*, the entire system), and not on partial results (*e.g.*, a subsystem) which may also be of value.

- Software Engineering researchers have been investigating similarity measurements [2, 3, 13] to quantify the level of agreement between pairs of clustering results. Researchers tend to focus on using these measurements to compare clustering results directly; however, we know of no work that tries to find common patterns in clustering results that are produced by a family of different clustering algorithms.

- The importance of evaluating clustering techniques has been investigated by Koschke and Eisenbarth [9]. Their paper describes several techniques for comparing the results of a clustering algorithm to a reference decomposition. The authors also describe a consensus-driven approach for creating a reference decomposition. However, their approach for establishing the reference decomposition is manual, requiring teams of software engineers.

To address some of the problems mentioned above, we propose a framework that supports a process for automatically creating views of a system's structure that are based on common patterns produced by various clustering algorithms. The initial step in our process clusters a system many times using different clustering algorithms. For each clustering run, all of the modules that appear in the same cluster are recorded. Using this information our framework presents consolidated views of the system structure to the user. By using a collection of clustering algorithms we "average" the different clustering results and present views based on common agreement across the various clustering algorithms. In the next section we introduce our framework, which we have implemented and made

[1] This problem is interesting and has received very little formal attention by researchers. We are investigating how dynamic program behavior can be integrated into our clustering techniques.

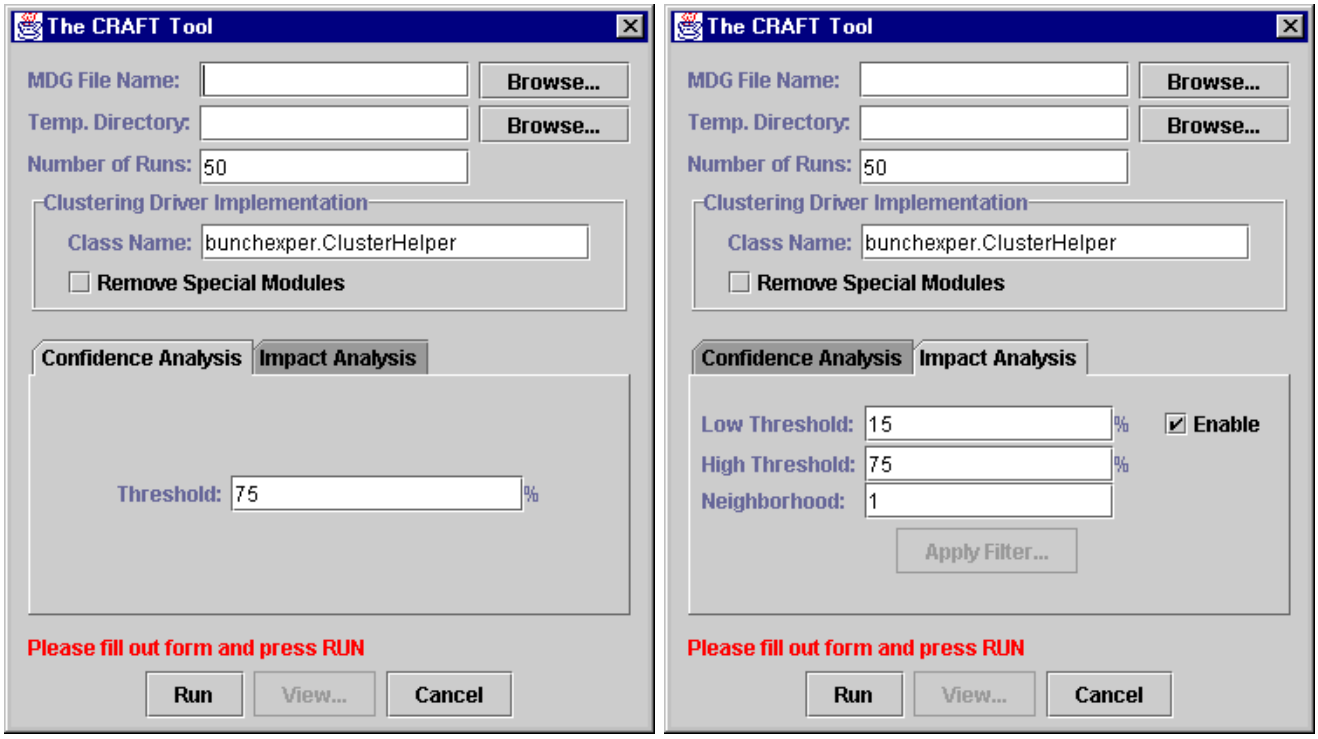

Figure 1. The User Interface of the CRAFT Tool

available to researchers on the world wide web at http://serg.mcs.drexel.edu/bunch/CRAFT.

3. Our Framework - CRAFT

In Figure 1 we illustrate the user interface of our clustering analysis tool, which we call *CRAFT* (**C**lustering **R**esults **A**nalysis **F**ramework and **T**ools). The main window is organized into two sections. The section at the top of the window collects general parameters, and the section at the bottom of the window accepts thresholds that apply to a particular analysis service. Our tool currently supports two such services, *Confidence Analysis* and *Impact Analysis*, which are discussed later in this section.

The overall goal of the CRAFT framework is to expose common patterns in the results produced by different clustering algorithms. By highlighting the common patterns, we gain confidence that agreement across a collection of clustering algorithms is likely to reflect the underlying system structure. We are also interested in identifying modules/classes that tend to drift across many clusters, as modifications to these modules/classes may have a large impact on the overall system structure.

As shown in Figure 2, the CRAFT architecture consists of the following subsystems:

- **The User Interface:** CRAFT obtains information from the user to guide the collection, processing and visualization of clustering results.
- **Clustering Services:** The actual clustering process is externalized to a dynamically loadable clustering driver. The clustering driver is responsible for partitioning a graph that represents the components and dependencies of a software system into a set of non-overlapping clusters. The CRAFT framework invokes the clustering driver multiple times based on the value provided by the user in the *Number of Runs* entry field.
- **Data Analysis Services:** The results of the clustering activity are stored in a relational database. For each clustering run, every pair of modules in the system is recorded to indicate if the modules in the pair are in the same or different clusters. The repository of clustering results supports two types of analyses: *Confidence Analysis* and *Impact Analysis*. Confidence Analysis produces a decomposition of the system based on common trends in the clustering data. Impact Analysis examines each module in the system to determine how it relates to all of the other modules in the system with respect to its placement into clusters. Confidence Analysis helps users gain confidence in a

clustering result when no reference decomposition exists. Impact Analysis helps users perform local analyses, which is much simpler than analyzing the entire decomposition of a system.

- **Visualization:** Once the results of the clustering analysis have been produced, our tool presents the results in diagrammatic form.

We next present a small example illustrating the capabilities of the CRAFT framework followed by a detailed description of the above mentioned services.

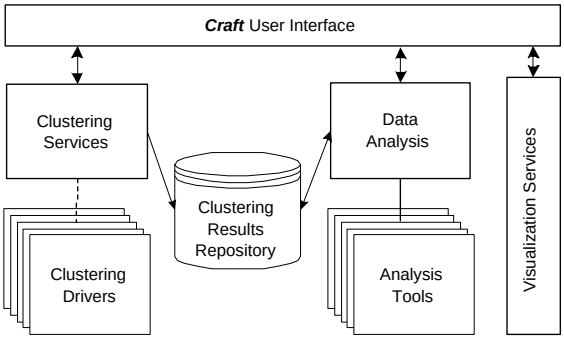

Figure 2. The Architecture of the CRAFT Tool

3.1. A Small Example

In Figure 3 we illustrate a small example to help explain the usage of the CRAFT framework. The Module Dependency Graph (MDG), which is illustrated at the upper-left corner of the figure, shows a system consisting of 5 modules $\{M1, M2, M3, M4, M5\}$ with 6 dependencies between the modules. The MDG is a generic, language independent representation of the structure of the system's source code components. This representation includes all of the modules (classes) in the system and the set of dependencies that exist between the modules (classes).

Intuitively, based on the dependencies between the modules, we expect that $\{M1, M2\}$ and $\{M4, M5\}$ should appear in the same cluster. We also expect that module $M3$ is equally likely to appear in the $\{M1, M2\}$ or $\{M4, M5\}$ cluster since it is isomorphic to both of them.

We executed CRAFT with a setting that clustered the example system 100 times. The data from the clustering results repository is shown on the bottom-left corner of Figure 3. Each relation in the repository indicates the frequency that a pair of modules appears in the same cluster. Visual inspection of this data confirms that modules $M1$ and $M2$, along with modules $M4$ and $M5$ appear in the same cluster 100% of the time. The data also shows that module $M3$ appears in the $\{M1, M2\}$ cluster 57% of the time and in the $\{M4, M5\}$ cluster 43% of the time.

The results of the Impact Analysis are shown in the center of Figure 3. To conserve space, we have configured the CRAFT user interface to show only the modules that exceed an upper threshold (as shown by the $\uparrow$ icon) that was selected by the user. In Section 3.4.2 we describe additional features of the Impact Analysis service inclusive of the thresholds that can be specified by the user. The Impact Analysis results clearly show that modules $M1$ and $M2$ always appear together, modules $M4$ and $M5$ always appear together, and that module $M3$ does not appear in any particular cluster consistently (the icon for $M3$ cannot be expanded).

On the right-side of Figure 3 we show the results of the Confidence Analysis. The partition of the system shown in the Confidence Analysis result is consistent with our intuitive decomposition, as well as the view produced by the Impact Analysis service. Note that the edges shown in Figure 3 do not depict dependencies in the system being studied. The edges instead represent the frequency percentage that the two modules appear in the same cluster based on all of the clustering runs. The cluster containing only module $M3$ does not necessarily indicate that this module appears alone in any of the clustering results. In the Confidence Analysis visualization, singleton clusters should be interpreted as modules that do not appear in any particular cluster consistently.

Given such a small example, the value of having multiple views of the clustering results data is not apparent. In Section 4, where we present the results of a case study on a non-trivial example, the benefits of the two views becomes more obvious. Now that a simple example has been has been presented, we return to discussing the architecture and services of CRAFT.

3.2. The User Interface

The user interface of our tool, which is shown in Figure 1, collects information that is necessary to analyze the results of a collection of clustering algorithms. The key information collected on the user interface is the text file name of the MDG, the number of times the clustering step will be performed (described in Section 3.3), the name of a Java Bean that will perform the clustering activities, and the thresholds that are used by the data analysis and visualization services. The tab at the bottom of the window is used to select the analysis service to be performed, and to collect the parameters associated with each analysis service.

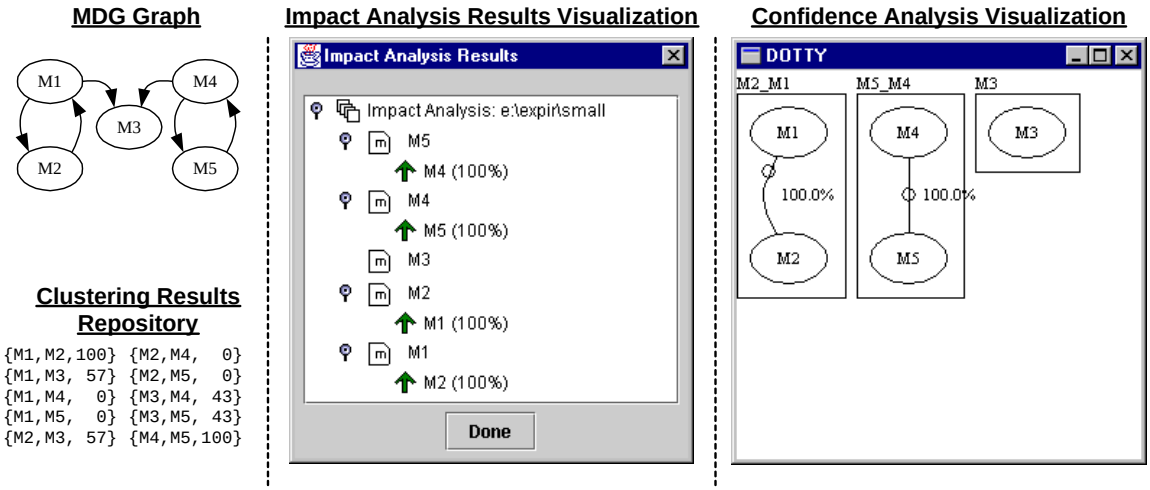

Figure 3. A Small Example

3.3. The Clustering Service

Prior to using our tool, a clustering driver class must be created and packaged as a Java Bean. The responsibility of this driver is to accept as input: (a) the file name of the MDG, (b) the name of an output file that the clustering driver is expected to produce, (c) the current run number and (d) the collection of the parameters specified on the CRAFT user interface. The clustering driver encapsulates the algorithm(s) used to cluster the system. The input file is provided in our *MDG* file format, and the output file must adhere to our *SIL* file format. Furthermore, the clustering driver must extend the `AnalysisTool.IClusterHelper` interface, which is included in the distribution of our tool. The clustering analysis tool, programmer and user documentation, along with associated file format specifications can be obtained online at the Drexel Univiersity Software Engineering Research Group (SERG) web page at http://serg.mcs.drexel.edu/bunch/CRAFT.

The need to create an external clustering driver is consistent with our philosophy of developing tools that can be used by other researchers in the Software Engineering community. Our approach allows researchers to create clustering drivers in the Java programming language. Clustering tools not developed in Java can also be integrated into our framework using Java's JNI [8] capability to wrap tools developed in other programming languages. For the convenience of the user, we have created and included two drivers with our tool. Both drivers are based on algorithms supported by Bunch. Specifically:

- class `bunchexper.ClusterHelper`: This clustering driver uses the Bunch NAHC clustering algorithm [12, 11, 14]. All of the algorithms supported by Bunch use a randomized optimization approach to form clusters. Thus, repeated runs of Bunch, with the same algorithm, rarely produce the exact same result.

- class `bunchexper.ClusterHelperExt`: Based on the number of clustering runs specified by the user, this driver applies the Bunch NAHC clustering algorithm 25% of the time, the Bunch SAHC algorithm 25% of the time, our generic hill climbing algorithm 40% of the time, and our Genetic Algorithm [5] 10% of the time.

  The generic hill climbing algorithm can be tuned using configuration parameters to behave like any algorithm on the continuum between our NAHC and SAHC hill climbing algorithms [12]. Overall, 10 of the Bunch clustering algorithms are included in the `bunchexper.ClusterHelperExt` driver. In addition to the NAHC, SAHC and Genetic algorithms, 7 different configurations of the generic hill climbing algorithm are used.

3.4. The Data Analysis Service

Once the clustering activity finishes, the data associated with each clustering run is stored in the Clustering Results Repository. Next, the data is processed by our analysis tools to support the construction of useful views of the data. Our goal is to consolidate the diversity in the results produced by the set of clustering tools so that common patterns can be identified. This approach has shown that good decompositions can be produced in the presence of a set of clustering algorithms. Instead of relying on a benchmark decomposition to evaluate the result, the result produced by

CRAFT is likely to be good because the set of clustering algorithms "reached a consensus".

As mentioned above, the first step performed during data analysis is the consolidation of the data that has been saved in the clustering results repository. Let $\mathcal{S} = \{M_1, M_2, \ldots M_n\}$ be the set of all modules in the system. For each distinct pair of modules (M_i, M_j), where $1 \leq i, j \leq |\mathcal{S}|$, and $M_i \neq M_j$, we calculate the frequency (α) that this pair appears in the same cluster. Given that the clustering driver executes Π independent runs, the number of times that a pair of modules can appear in the same cluster is bounded by $0 \leq \alpha \leq \Pi$. The principle that guides our analysis is that the closer α is to Π, the more likely that the pair of modules belongs to the same cluster. During the first step of the data analysis, the consolidated data is saved in the Clustering Results Repository so that it can be used by our data analysis tools.

Given a system consisting of n modules that has been clustered Π times, let $\alpha_{i,j}$ be the number of times that Modules M_i and M_j appear in the same cluster. We also define $\mathcal{C}[m1, m2, \alpha]$ as the schema[2] for the data in the clustering repository that includes all of the $\{M_i, M_j, \alpha_{i,j}\}$ triples. Each $\{M_i, M_j, \alpha_{i,j}\}$ relation represents the number of times that Module M_i and Module M_j appear in the same cluster.

For convenience we also define $\mathcal{D}$ to be a view on $\mathcal{C}$ such that all of the triples in $\mathcal{D}$ are sorted in descending order based on $\mathcal{D}[\alpha]$.

Now that the above definitions have been presented, we turn our attention to describing the two data analysis services provided by CRAFT.

3.4.1 Confidence Analysis Tool

The Confidence Analysis Tool (CAT) processes the tuples in the Clustering Results Repository to produce a reference decomposition of a software system. Let β represent a user-defined threshold value, which is set on the user interface (see the left side of Figure 1). Given the ordered set of tuples in the Clustering Results Repository (*i.e.*, set $\mathcal{D}$), the user specified threshold β and the set of all of the modules in the system (*i.e.*, set $\mathcal{S}$), we create the reference decomposition by applying Algorithm 1.

As Algorithm 1 shows, the first step in the CAT process involves forming an initial cluster by taking the relation from $\mathcal{D}$ with the largest α value. The closure of the modules in this relation is then calculated and new modules are added to the cluster by searching for additional relations in $\mathcal{D}$ such that one of the modules in the triple is already contained in the newly formed cluster, and the α value exceeds the user defined threshold β.

[2]Given a relation c for the schema $\mathcal{C}$, we use the notation $c[m1, m2]$ to represent the projection of $m1$ and $m2$ on $\mathcal{C}$. For example, given the relation c ={parser,scanner,10}, $c[m1, m2]$ ={parser,scanner}.

Algorithm 1: Confidence Analysis Algorithm

ConfidenceAnalysis(Set: $\mathcal{D}$, $\mathcal{S}$; Threshold: β)
  Let $\mathcal{P}$ be a new partition of the system.
  foreach relation $d \in \mathcal{D}$ **sorted**
  decreasing by $d[\alpha]$ **where** $d[\alpha] \geq \beta$ **do**
    if ($d[M_1]$ **or** $d[M_2]$) **is in** $\mathcal{S}$ **then**
      1. Create a new cluster $\mathcal{C}$ and add it to partition $\mathcal{P}$. Add the modules from relation d ($d[M_1]$ and/or $d[M_2]$) to $\mathcal{C}$ that are in $\mathcal{S}$. Remove the modules that have just been added to the new cluster $\mathcal{C}$ from $\mathcal{S}$.
      2. CloseCluster: Search for additional relations $r \in \mathcal{D}$ that have $r[\alpha] \geq \beta$. Select the relation with the highest $r[\alpha]$ value such that one module from r is in $\mathcal{C}$ and the other module from r is in $\mathcal{S}$. Add the module from r that is in $\mathcal{S}$ to $\mathcal{C}$. Once added to $\mathcal{C}$, remove this module from $\mathcal{S}$. Repeat this process until no new modules can be added to $\mathcal{C}$.
    end
  end
  foreach module s **remaining in** $\mathcal{S}$ **do**
    Create a new cluster $\mathcal{C}$ and add it to $\mathcal{P}$.
    Add module s to cluster $\mathcal{C}$.
  end
return (**partition** $\mathcal{P}$)

When adding modules to clusters, care is taken so that each module in set $\mathcal{S}$ is assigned to no more than one cluster, otherwise we would not have a valid partition of the MDG. After the closure of the cluster is calculated, a new cluster is created and the process repeats. Eventually, all modules that appear in relations in $\mathcal{D}$ that have an α value that exceeds the user defined threshold β will be assigned to a cluster. In some cases, however, there may be modules for which no relation in $\mathcal{D}$ exists with a large enough α value to be assigned to a cluster. In these instances, for each remaining unassigned module in $\mathcal{S}$, a new cluster is created containing a single module. This condition is not an indication that the module belongs to a singleton cluster, but instead is meant to indicate that the module is somewhat "unstable" in its placement, as it tends not to get assigned to any particular cluster.

The final step performed by the CAT tool is to compare the decomposition that it produced with each of the decompositions that were generated during the clustering process. This is accomplished by measur-

ing the similarity between the result produced by the CAT tool with each result produced by the clustering algorithms. CRAFT currently provides the user with the average, standard deviation, minimum and maximum values of 3 similarity measurements. The similarity measurements used are Precision/Recall [2, 3], EdgeSim, and MeCl. The Precision/Recall and EdgeSim measurements evaluate the similarity of the clusters in two distinct decompositions of a system. MeCl is a distance similarity measurement that determines the number of logical operations required to convert one decomposition into the other. We discuss the EdgeSim and MeCl measurements in one of our other papers [13]. Using similarity measurements is a good "sanity check" to ensure that the information provided by CRAFT is representative of the individual clustering results. If the similarity measurements are high, we are confident that the results produced by CRAFT are good.

In this section we illustrated how the CAT tool can be used as a "reference decomposition generator". Such a tool is useful when a standard benchmark decomposition does not exist. We would also like to point out that the singleton clusters produced by the CAT tool provide useful information, as they identify modules that are not assigned to any particular cluster with regularity.

3.4.2 Impact Analysis Tool

The Impact Analysis Tool (IAT) helps developers understand the local impact of changing a module. This tool is appropriate for specialized analyses where the developer wants to quickly find the most closely related modules to a specific module in the system. The IAT helps developers understand which modules are most likely to be impacted by a change, which modules are least likely to be impacted by a change, and the modules for which the impact of a change is unknown.

Like the CAT tool, the IAT uses the data in the Clustering Results Repository. Given a module in the system, the Repository ($\mathcal{D}$) is queried to determine the set of related modules (a dependency exists in the MDG) and their associated α values. If the α value exceeds an upper threshold (γ), or is below a lower threshold (δ) then a we can generally infer if the module is in the same or in a different cluster, respectively. However, if we find that $\gamma < \alpha < \delta$ we can infer that the module is somewhat, but not very strongly, dependant on the other module. The thresholds γ and δ are set on the CRAFT user interface (see the right side of Figure 1). The detailed behavior of IAT is shown in Algorithm 2.

CRAFT requires the user to enter an upper threshold (γ), and an optional lower threshold (δ). By default, the IAT tool produces results for all of the modules in the system. However, the user may select which modules will be included in the visualized results. To limit the modules considered by the IAT tool, the user presses the *Apply Filter* button on the CRAFT user interface (Figure 1). Once the IAT tool finishes processing the data in the Clustering Results Repository, the output is visualized using a hierarchial presentation of the data. Figure 4 illustrates sample output for the IAT tool. Notice how an up arrow ($\uparrow$) is associated with modules that are above the upper threshold, and a down arrow ($\downarrow$) is associated with modules that are below the lower threshold. Beside each module name is the normalized α value ($\alpha/\Pi \times 100$) associated with the modules relation to its parent. For example, in Figure 4, we see that module `TimerQueue` is very associated with module `JRootPane` (100%) but not with module `MultiToolTipUI` (14%).

Algorithm 2: Impact Analysis Algorithm

ImpactAnalysis(Set: $\mathcal{D}$, $\mathcal{S}$; Threshold: γ, δ)
  Let $\mathcal{R}$ be the root of the results tree.
  foreach module $s \in \mathcal{S}$ **do**
    Let $\mathcal{Q} = \mathcal{S} - s$.
    Let $\mathcal{I}_s$ be a node that is inserted into tree $\mathcal{R}$ to represent module s.
    foreach $q \in \mathcal{Q}$ **do**
      Find the relation $d \in \mathcal{D}$ **where** $d[M_1, M_2]$ contains modules q and s.
      switch $d[\alpha]$ **do**
        case $(d[\alpha] \leq \gamma)$: $\mathcal{I}_s.Below \leftarrow \mathcal{I}_s.Below \cup q$
        case $(d[\alpha] \geq \delta)$: $\mathcal{I}_s.Above \leftarrow \mathcal{I}_s.Above \cup q$
        otherwise $\mathcal{I}_s.Neither \leftarrow \mathcal{I}_s.Neither \cup q$
      end
    end
  end
return $(\mathcal{R})$

The IAT tool also allows the user to select a *neighborhood*. By default the neighborhood is 1, however the user may set this to a maximum value that is equal to to the number of modules in the system. If a neighborhood value greater than 1 is provided, the IAT tool expands (if necessary) each node in the IAT results window to show modules that are transitively related to the selected module.

3.5. The Visualization Service

Once the data analysis is performed, the results are visualized. The actual visualization of the results de-

pends on whether the Confidence or the Impact analysis was performed in the data analysis step. For the CAT tool, the result is shown using the dotty [6] tool. For the IAT tool, the visualization of the results is presented hierarchically in a window that can be navigated by the user (see Figures 3 and 4).

4. Case Study

In this section we present a case study to illustrate the benefits of the CRAFT framework. Although we used one of the clustering drivers that was described in Section 3.3, we hope that other researchers will build clustering drivers to integrate additional clustering algorithms into CRAFT.

The system that we analyzed is the Swing [18] class library, which is part of the Java Developers Kit(JDK). The Swing library consists of 413 classes that have 1513 dependencies between them. The results of the case study are based on clustering Swing 100 times using our `bunchexper.ClusterHelperExt` clustering driver. We chose the `bunchexper.ClusterHelperExt` clustering driver, which we described in Section 3.3, because it uses a family of 10 (*i.e.*, NAHC, SAHC, 7 variations of our generic hill climbing algorithm, and a genetic algorithm) clustering algorithms that have been integrated into Bunch.

Similarity Measurement	Avg	Min	Max	Stdev
MeCl	96.5%	92.1%	100.0%	0.17
EdgeSim	93.1%	89.3%	98.6%	0.80
Precision/Recall	82.5%	54.8%	91.4%	1.97

Table 1. Similarity Results for CAT Test

The results of the Confidence Analysis Test are illustrated in Figure 5. In Table 1 we show the similarity measurements [13] indicating how close the clusters produced in the Confidence Analysis Test are to each of the 100 clustering results. Because of space restrictions, the CAT result displayed in Figure 5 only shows a part of the Swing package. The entire decomposition for Swing can be viewed online from the CRAFT webpage (http://serg.mcs.drexel.edu/bunch/CRAFT). In Figure 4 we illustrate the results of the IAT test. Several interesting conclusions can be made from the results of this case study:

- Many of the edges shown in the CAT visualization have large values (shown as labels on the edges). Recall that the edges in the CAT result do not represent the dependencies in the system, but are used to indicate the frequency that pairs of modules appear in the same cluster. Since many of the edge labels are high, the results indicate that there is general agreement across the 100 clustering results.

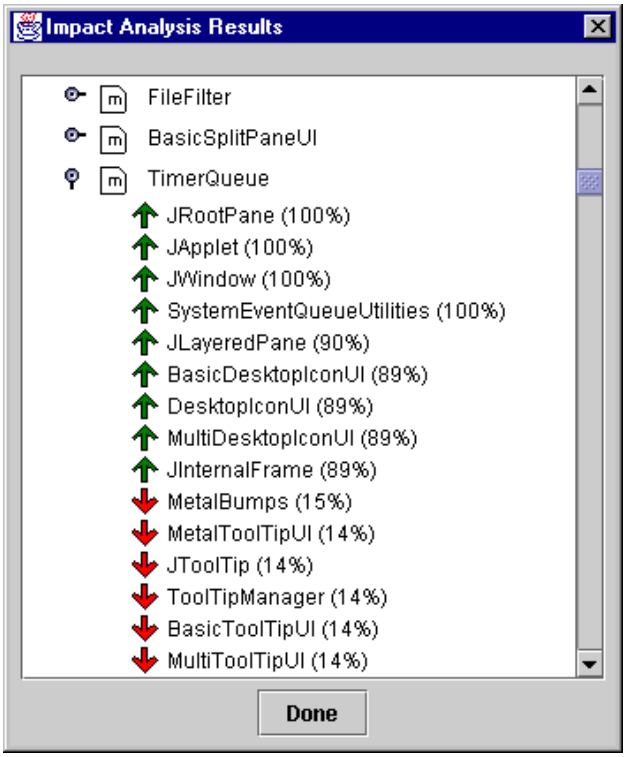

Figure 4. Swing: Impact Analysis Results

- The similarity measurements shown in Table 1 are high which is a good indication that the CAT results are representative of the underlying system structure.

System Name	Modules/ Classes	Edges	Execution Time (sec.)
ispell	24	103	12
rcs	29	163	13
bison	37	179	18
grappa	86	295	56
Swing	413	1513	730

Table 2. Performance of the CRAFT Tool

- The Impact and Confidence Analysis results are based on the same set of data. Figures 4 and 5 illustrate how both visualizations of this data can be useful. The CAT result in Figure 5 shows a partial reference decomposition for Swing that was produced by CRAFT. This result is helpful for gaining some intuition into Swing's overall structure. However, the obvious complexity

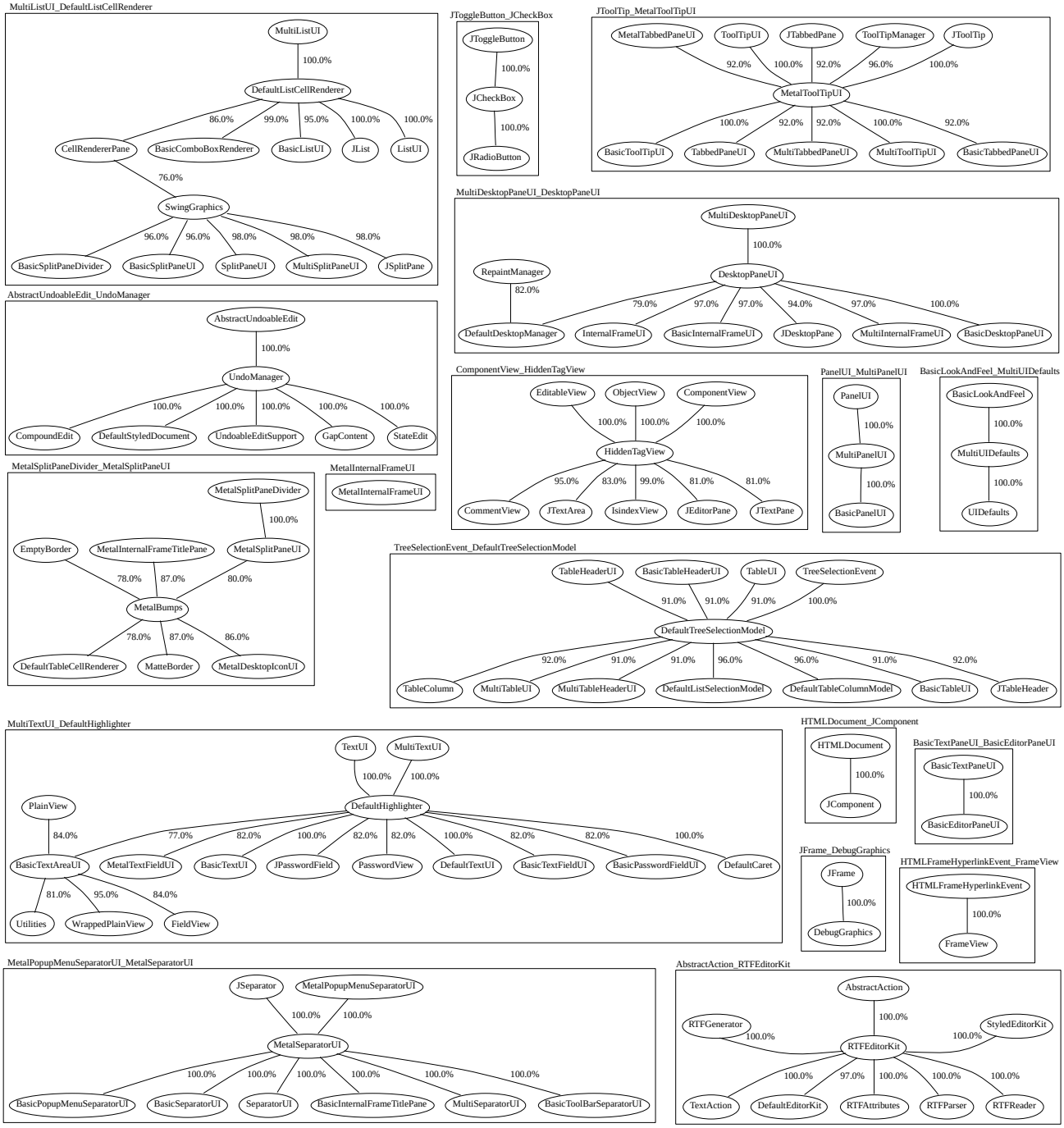

Figure 5. Confidence Analysis Result: A Partial View of Swing's Decomposition

of the CAT results may overwhelm developers who are trying to understand the potential impact associated with modifying a particular class within the Swing package. The IAT results address this concern by highlighting the modules that are most likely to be impacted by a change to a single module. For example, Figure 4 illustrates that the `JRootPane`, `JApplet`, `JWindow` and `SystemEventQueueUtilities` classes always appear in the same cluster as the `TimerQueue` class.

- The performance of CRAFT seems to be good for medium to large systems (up to 1000 modules). However, we have not tried the CRAFT tool on very large systems. In Table 2 we show the performance of the CRAFT tool for systems of vari-

ous sizes that were clustered 100 times using the `bunchexpir.ClusterHelperExt` clustering driver.

5. Conclusions

In this paper we investigated how the diversity in results produced by different clustering algorithms can be used to generate a reference decomposition of a software system. Koschke and Eisenbarth [9] state that a reference decomposition is necessary in order to compare the strengths and weaknesses of individual clustering algorithms. From the perspective of a researcher we agree with this claim, but we also argue that a reference decomposition is useful to software practitioners who are trying to gain confidence in the results produced by a clustering algorithm.

We also presented the CRAFT framework, which can generate a reference decomposition automatically. Manual construction of a reference decomposition is tedious, but the results will tend to be good because knowledge from the designers of the system is used. However, a formal process [9] is needed to guide the manual construction of the reference decomposition so that personal biases about the system structure are minimized. As future work we would like to compare the CAT results from CRAFT to manually produced decompositions that have been created by other researchers.

We have also shown how the process of deriving the reference decomposition can lead to other useful results such as Impact Analysis.

Finally, the CRAFT framework can be downloaded over the Internet, and is capable of being extended by other researchers. We hope that this work will trigger additional interest in techniques to evaluate software clustering results.

6. Acknowledgements

This research is sponsored by grants CCR-9733569 and CISE-9986105 from the National Science Foundation (NSF). Additional support was provided by the research laboratories of AT&T, and Sun Mircosystems.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, U.S. Government, AT&T, or Sun Microsystems.

We would like to thank Banning Cohen for his help with the illustrations.

References

[1] N. Anquetil. A comparison of graphis of concept for reverse engineering. In *Proc. Intl. Workshop on Program Comprehension*, 2000.

[2] N. Anquetil, C. Fourrier, and T. Lethbridge. Experiments with hierarchical clustering algorithms as software remodularization methods. In *Proc. Working Conf. on Reverse Engineering*, 1999.

[3] N. Anquetil and T. Lethbridge. Recovering software architecture from the names of source files. In *Proc. Working Conf. on Reverse Engineering*, 1999.

[4] S. Choi and W. Scacchi. Extracting and restructuring the design of large systems. In *IEEE Software*, pages 66–71, 1999.

[5] D. Doval, S. Mancoridis, and B. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *Proceedings of Software Technology and Engineering Practice*, 1999.

[6] E. Gansner, E. Koutsofios, S. North, and K. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, Mar. 1993.

[7] D. Hutchens and R. Basili. System Structure Analysis: Clustering with Data Bindings. *IEEE Transactions on Software Engineering*, 11:749–757, Aug. 1985.

[8] Javasoft. http://www.javasoft.com.

[9] R. Koschke and T. Eisenbarth. A framework for experimental evaluation of clustering techniques. In *Proc. Intl. Workshop on Program Comprehension*, 2000.

[10] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proc. International Conference on Software Engineering*, 1997.

[11] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of International Conference of Software Maintenance*, Aug. 1999.

[12] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proc. 6th Intl. Workshop on Program Comprehension*, 1998.

[13] B. S. Mitchell and S. Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *Proceedings of International Conference of Software Maintenance*, 2001.

[14] B. S. Mitchell, M. Traverso, and S. Mancoridis. An architecture for distributing the computation of software clustering algorithms. In *The Working IEEE/IFIP Conference on Software Architecture*, Aug. 2001.

[15] H. Müller, M. Orgun, S. Tilley, and J. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5:181–204, 1993.

[16] R. Schwanke. An intelligent tool for re-engineering software modularity. In *Proc. 13th Intl. Conf. Software Engineering*, May 1991.

[17] R. Schwanke and S. Hanson. Using Neural Networks to Modularize Software. *Machine Learning*, 15:137–168, 1998.

[18] Javasoft Swing Libraries: Java Foundation Classes. http://www.javasoft.com/products/jfc/index.html.

[19] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proc. International Conference on Software Engineering*, Aug. 1999.