MoJo : A Distance Metric for Software Clusterings

Vassilios Tzerpos
University of Toronto
Toronto, Ontario, CANADA
vtzer@cs.toronto.edu

R.C. Holt
University of Waterloo
Waterloo, Ontario, CANADA
holt@plg2.math.uwaterloo.ca

Abstract

The software clustering problem has attracted much attention recently, since it is an integral part of the process of reverse engineering large software systems. A key problem in this research is the difficulty in comparing different approaches in an objective fashion. In this paper, we present a metric that can be used in evaluating the similarity of two different decompositions of a software system.

Our metric calculates a distance between two partitions of the same set of software resources. We begin by introducing the model we use. Consequently, we present a heuristic algorithm that calculates the distance in an efficient fashion. Finally, we discuss some experiments that showcase the performance of the algorithm and the effectiveness of the metric.

1 Introduction

The research field of reverse engineering is concerned with the process of discovering the structure of existing software systems in an effort to understand, redocument and possibly modify these systems. The number of applications for reverse engineering processes has increased dramatically in the recent years. Some of the reasons for this phenomenon are the need to port existing systems to new hardware platforms; the modification of the system's requirements; problems such as the Y2K bug; and the obsoleteness of the system's documentation.

Software clustering techniques are commonly employed in order to aid the reverse engineering process. These techniques attempt to discover natural groupings of large numbers of software objects, such as the set of all files comprising a software system. Their goal is to break the problem of understanding a large piece of software into smaller, easier to manage subproblems, thus reducing the amount of effort involved in recovering the structure of the given software system.

A variety of software clustering approaches has been presented in the literature. Each of these approaches looks at the software clustering problem from a different angle, by either trying to compute a measure of similarity between software objects[11]; deducing clusters from file and procedure names[1]; utilizing the connectivity between software objects[4, 9, 2]; or looking at the problem at hand as an optimization problem[6]. All of these techniques seem to be successful on a number of examples. However, not only is there no approach that is widely recognized as superior, but it is also hard to compare the effectiveness of different approaches.

One reason for this situation is that most software clustering techniques are validated by applying them to example systems that are either proprietary or too small to draw any educated conclusions from. Even if we could test these techniques on the same large system,[1] how will we able to evaluate the results? The evaluation criterion used by most methods is developer feedback, in which system designers and seasoned developers offer their opinion on how much the obtained clustering reflects the actual breakdown of their system. This method however would be impractical when several techniques are being tested. Besides, a large degree of subjectivity is inherent in such an evaluation mechanism. Thus, it becomes clear that an objective means of measuring the effectiveness of a clustering approach is required.

In this paper, we present a metric that gauges how "close" two different clusterings are. In other words, the metric accepts two distinct decompositions of the resources of a software system as input, and outputs a number that represents a measure of the "distance" between them. Such a metric will be helpful in our effort to compare the effectiveness of different clustering algorithms in the following ways:

1. Many of the approaches presented in the literature are parameterized, i.e. they require the fine tuning of the value of certain parameters in order to be effective [6, 8, 5]. Using our metric we can determine how drastic an

[1] Popular open-source systems such as Netscape or Linux would be good candidates.

effect a change of value has on the derived clustering, and thus assess the importance of each parameter.

2. An important feature of clustering algorithms that hasn't attracted as much attention as it should so far, is their stability, i.e. how is the output of the algorithm (the obtained partition) affected when its input (the software system) is slightly modified. Researching this issue can be aided by our metric. It will be easier to determine the effect small changes in the software system have on the output of a clustering algorithm.

3. In the cases of software systems for which an "authoritative" clustering already exists (as is the case with systems that have been clustered manually by their architect), this metric will help us determine how close a given algorithm can come to reproducing it.

The structure of the rest of the paper is the following. Section 2 presents the model we use in order to measure similarity between two partitions. Section 3 describes a heuristic algorithm that approximates the value of the metric efficiently. Section 4 presents some experiments that we have performed using an implementation of the algorithm presented in section 3. These experiments showcase the performance of the algorithm and the effectiveness of the metric. Finally, section 5 concludes the paper.

2 The MoJo Model

An obvious way to compare the effectiveness of different clustering techniques is to apply them on a number of large systems and compare the results. However, this is not easy to do because:

- Large systems have a large number of resources. Viewing a partition of these resources, even with an interactive tool such as the Software Bookshelf[3], can be time-consuming. This makes attempting to compare a number of such partitions almost impractical.

- The text output of most software clustering approaches is more compact, but unsuitable for humans to read.

Therefore, it is clear that we need an automatic way to compare two partitions of the same large set. We propose a metric called MoJo that measures the "distance" between two partitions, i.e. the smaller the value of the metric, the more similar the two partitions are, and vice versa. In that sense, MoJo is a dissimilarity measure.

In order to compute this distance, MoJo uses an idea that has been used in other disciplines[2] that employ clustering techniques[10]: it counts the minimum number of operations (such as moving a resource from one cluster to another, joining two clusters etc.) one needs to perform in order to transform one partition to the other.

The MoJo metric is based on two operations, both of which are assigned a weight of 1. The simplest operation is that of moving a resource from one cluster to another. The other operation, that of joining two clusters, is also assigned a weight of 1 for the following reasons:

- From a mathematical point of view, it is an operation that can only be done in one way (as opposed to the operation of splitting a cluster of size n into two, which can be done in $2^{n-1} - 1$ ways).

- From a software clustering point of view, the fact that we need to join two clusters means that we performed more detailed clustering than required.

Splitting a cluster into two however, must be assigned a larger weight, as it is an operation that can be done in an exponential number of ways. MoJo overcomes this problem by disallowing the *split* operation (and effectively assigning it a weight equal to the cardinality of the smaller of the two new clusters, since a *split* operation can be simulated by a series of *move* operations).

To summarize, MoJo allows only two operations (it is from these two operations that MoJo gets its name):

1. *move* : **Mo**ving a resource from one cluster to another (that includes moving a resource into a previously non-existent cluster, thus creating a new cluster of cardinality 1).

2. *join* : **Jo**ining two clusters into one, thus reducing the number of clusters by 1.

Let us denote by $mno(A, B)$ the minimum number of operations to transform partition A into partition B. The value of our metric is defined as:

$$MoJo(A, B) = min(mno(A, B), mno(B, A))$$

It is interesting to note that it does not necessarily hold that $mno(A, B) = mno(B, A)$. For example, if partition A is $\{\{r_1, r_2, r_3\}, \{r_4, r_5, r_6\}\}$ and B is $\{\{r_1, r_2, r_3, r_1, r_2, r_3\}\}$, the two clusters of partition A can be joined in a simple operation. As a result, $mno(A, B) = 1$. However, $mno(B, A) = 3$ because three resources must be moved to split B's single cluster.

In the next section, we present a heuristic algorithm that approximates the value of our metric in an efficient manner.

[2] Examples include psychology, biology, statistics, social sciences, and various forms of engineering.

3 The heuristic algorithm

Finding the exact value of $MoJo(A, B)$, where A and B are two partitions of the same set of objects S, can be time-consuming when S is large. For this reason, we have implemented a heuristic algorithm that computes an approximation to the value of $MoJo(A, B)$ in a reasonable amount of time. Experiments using this implementation are presented in section 4. In this section we present the heuristic algorithm used.

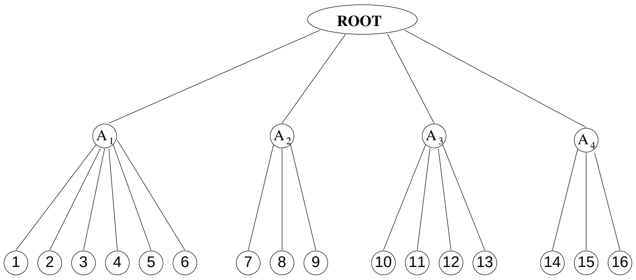

Figure 1. The containment tree of partition A.

Our algorithm computes an approximation to $mno(A, B)$. From that, we can compute an approximation to $MoJo(A, B)$ as shown in the previous section. Also, our algorithm assumes that A and B have a containment tree height of value 3, i.e. all the clusters are contained in a generic root, and all the objects are contained in these clusters (see figure 1). In other words, A and B are non-recursive partitions. It is easy to show that partitions of larger containment tree height can be transformed into partitions such as the one in figure 1 by inducing all objects to the first level clusters.[3]

Let $A_i, 1 \leq i \leq l$ be the clusters of partition A (an example with 16 objects is shown in figure 2), and $B_j, 1 \leq j \leq m$ be the clusters of partition B (a corresponding example is shown in figure 3).

Each object s in partition A is assigned a tag $T_j, 1 \leq j \leq m$ depending on which cluster s belongs to in partition B. For example, object ① is assigned tag T_1 since it belongs in cluster B_1, and object ⑤ is assigned tag T_2 since it belongs in B_2. The tags are shown in figure 2 as a rectangle next to each object (only the subscripts are shown to avoid cluttering, i.e. tag T_2 is shown as $\boxed{\mathbf{2}}$).

Let us denote by $A_i(T_j)$ the set of objects that are contained in cluster A_i and are assigned the tag T_j. For example, $A_1(T_1)$ contains objects ①, ②, ③, and ④, while $A_4(T_3)$ contains objects ⑮ and ⑯.

Our algorithm considers only the tags of each object from now on. To simplify things, we present partition A including only the tags of each object in figure 4.

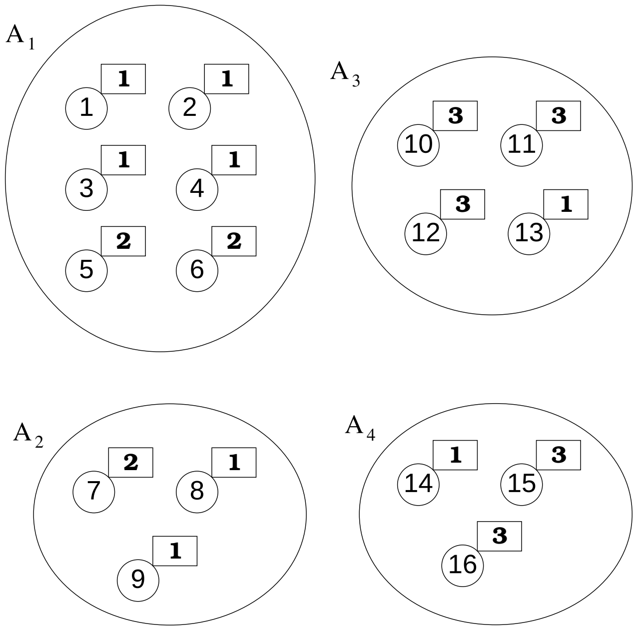

Figure 2. Example partition A. Objects are round, while tags are rectangular.

We now introduce the notion of *center cluster*. Each cluster B_j in partition B is assigned a center cluster A_i in partition A, such that $|A_i(T_j)| \geq |A_k(T_j)|, 1 \leq k \leq l$. If more than one of the clusters of A fit this definition, one of them is chosen arbitrarily. We will denote the center cluster of cluster B_j by CC_j. For example, we have that $CC_3 = A_3$, since cluster A_3 contains the largest number of objects tagged with T_3.

The intuition behind center clusters is that during our effort to transform A to B we will attempt to transform CC_j (which is a cluster in partition A) to B_j (a cluster in partition B). Therefore, it is necessary that each A_i is the center clus-

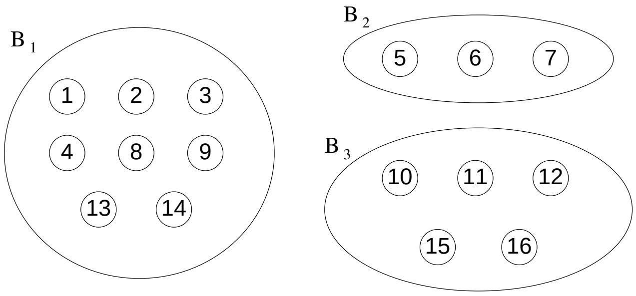

Figure 3. Example partition B. This partition is only used in order to assign the appropriate tags to the objects in partition A (figure 2).

[3]Clusters of any level can actually be used for this induction.

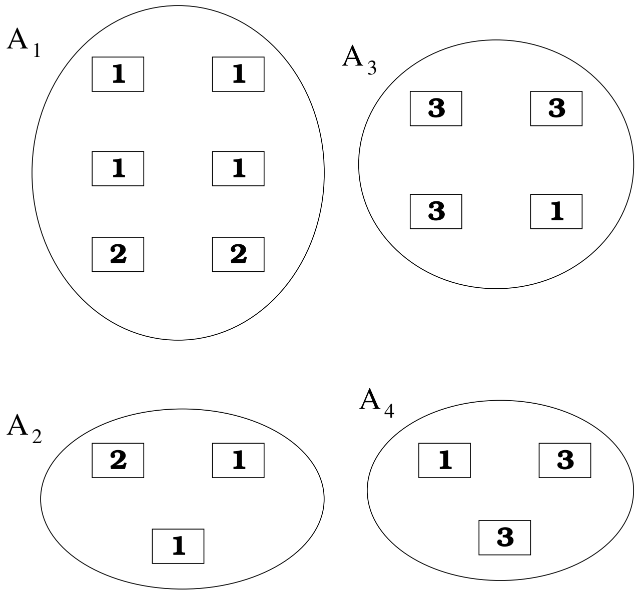

Figure 4. Partition A containing only tags.

ter of at most one B_j. However, the aforementioned definition of center clusters can result in the same cluster A_i being the center cluster of more than one clusters in B (in our example, we would have $CC_1 = CC_2 = A_1$, since cluster A_1 contains the largest subsets of $\boxed{1}$s and $\boxed{2}$s). As a result, we expand this definition by repeating the following procedure (shown as indented text) until all center clusters are distinct or no further improvement can be made. The intuition behind it is that, if we have $CC_x = CC_y = A_z$, we try to find the second largest subsets of $\boxed{x}$s and $\boxed{y}$s (in our example, A_2 contains the second largest subsets of $\boxed{1}$s and $\boxed{2}$s, so this procedure will attempt to make A_2 the center cluster of either B_1 or B_2).

> Let us assume that we have integers x, y, and z, such that $1 \leq x, y \leq m$, $1 \leq z \leq l$, and $CC_x = CC_y = A_z$ (in our example we would have $x = 1$, $y = 2$, $z = 1$). We also define sets $\Phi_k, 1 \leq k \leq m$ that are initially empty (these sets are used as markers, so that we do not consider the same A_i more than once). We compute the following values:
>
> $$x_1 = |A_z(T_x)|$$
>
> $$y_1 = |A_z(T_y)|$$
>
> $$x_2 = |A_f(T_x)|, \text{ such that } |A_f(T_x)| \geq |A_i(T_x)|, \quad 1 \leq i \leq k, i \neq z, i \notin \Phi_x$$
>
> $$y_2 = |A_g(T_y)|, \text{ such that } |A_g(T_y)| \geq |A_j(T_y)|, \quad 1 \leq j \leq k, j \neq z, i \notin \Phi_y$$
>
> If both $x_2 = 0$ and $y_2 = 0$, then no other A_i can be considered to be the center cluster of either B_x or B_y.
>
> If only $x_2 = 0$, then we assign $CC_y = A_g$ and add z to Φ_y.
>
> If only $y_2 = 0$, then we assign $CC_x = A_f$ and add z to Φ_x.
>
> Otherwise, if $x_1 + y_2 > x_2 + y_1$, then we assign $CC_y = A_g$ and add z to Φ_y, else we assign $CC_x = A_f$ and add z to Φ_x.
>
> In our example, we have $x_1 = 4$, $y_1 = 2$, $x_2 = 2$, and $y_2 = 1$, while $A_f = A_2$ and $A_g = A_2$. According to the above, we will now assign $CC_2 = A_2$.

It is still possible that there exist non-distinct center clusters after the end of the above procedure (this will surely happen if $|A| < |B|$). In this case, our algorithm begins the transformation by "splitting" these clusters into two, so that one of the two new clusters contains only the objects tagged with T_x, where $|A_z(T_x)| \leq |A_z(T_y)|$. This splitting is of course simulated as a series of *move* operations, and the algorithm adds their number in the current value of $mno(A, B)$.

At this point, a simple series of operations that accomplishes the transformation of partition A to partition B is one that moves all objects tagged with T_j into CC_j. However, in this way we do not benefit from the fact that joining two clusters has a weight of only 1. As a result, our algorithm will attempt to perform any "profitable" *join* operations before resorting to *move* operations.

In order to decide what a profitable *join* operation is, let us consider two clusters $\mathbb{X} = CC_x$ and $\mathbb{Y}$. If $\mathbb{Y}$ is not a center cluster, then it is profitable to join the two clusters if $\mathbb{Y}(T_x) > 1$. If $\mathbb{Y}$ is a center cluster, such that $\mathbb{Y} = CC_y$, then we define the following values:

$$A = |CC_x| \qquad \Gamma = |CC_y|$$

$$\alpha = |CC_x(T_x)| \qquad \gamma = |CC_y(T_y)|$$

$$\beta = |CC_y(T_x)| \qquad \delta = |CC_x(T_y)|$$

Let us assume that $\beta \geq \delta$ (if not, we can interchange clusters $\mathbb{X}$ and $\mathbb{Y}$). It is easy to show that transforming CC_x to B_x and CC_y to B_y by using only *move* operations will require a cost of $A - \alpha + \Gamma - \gamma$. At the same time, if we first join these two clusters into a new cluster $\mathbb{Z}$, and then proceed to transform to B_x and B_y (by moving everything except for $\mathbb{Z}(T_x)$ out of $\mathbb{Z}$, and creating a new cluster for $\mathbb{Z}(T_y)$), our cost will be $1 + A - \alpha + \Gamma - \beta$. Therefore, such a *join* operation is profitable only if $A - \alpha + \Gamma - \gamma < 1 + A - \alpha + \Gamma - \beta \Rightarrow 1 - \beta < -\gamma \Rightarrow \beta > \gamma + 1$.

As a result, our algorithm determines if such profitable *join* operations exist by examining all possible pairs of clusters. In order to achieve better performance, it starts with the cluster M that contains the largest set $M(T_i)$, such that if $M = CC_j$, we have $i \neq j$. It continues with the cluster containing the second largest such set etc. In this way, we minimize the number of *move* operations that will have to be performed.

In our example, it is profitable to join clusters A_3 and A_4 (in that way we move both objects tagged with T_3 into $A_3 = CC_3$ in one operation).

Finally, when no more profitable *join* operations remain, our algorithm moves all objects tagged with T_j into CC_j. In our example, there are 6 such *move* operations to be performed. As a result, we have $mno(A, B) = 7$ (this happens to be the exact value of $MoJo(A, B)$ as well).

In the next section, we present some experiments conducted using an implementation of this algorithm.

4 The experiments

The algorithm described in the previous section has been implemented. The implementation has been used to conduct several experiments. In this section we present some of these experiments that showcase the algorithm's efficiency as well as the usefulness of a metric such as MoJo.

4.1 Performance evaluation

The first experiment demonstrates that the value of the metric can be computed in a reasonable amount of time and that its computational complexity is no worse than linear with respect to the number of objects contained in each partition. We utilized two sources of data:

1. *Random data*. We have devised a clustering generator that, given an integer n creates a clustering of as many objects (the number of clusters is a random number between 5 and $\lfloor \frac{n}{5} \rfloor$. Our algorithm is then used to compute the distance between two such clusterings.

2. *Field data*. Our research team is associated with two large software systems. These systems have been clustered manually by domain experts. We are currently experimenting on these two systems with our own clustering algorithm.[4] We used our heuristic algorithm to compute the distance between the manual clustering and the one obtained from the clustering algorithm.

Table 1 summarizes our findings. Note that the values presented for the random data are average values obtained after creating 20 different clusterings and computing the distance between each pair (190 comparisons).

Type of data	Objects	Time
Random	200	0.17 sec
Random	600	0.7 sec
Field	939	0.67 sec
Random	1000	3.38 sec
Random	1400	4.85 sec
Random	1800	6.79 sec
Field	2097	6.93 sec
Random	2200	8.84 sec

Table 1. Summary of results.

Another interesting observation is that our algorithm seems to perform better with field data rather than random data. One of the reasons for this, is the fact that field data have similarities in their decompositions, something that our heuristic exploits in order to run faster.

4.2 Clustering algorithm stability

This experiment shows that our metric can be used to evaluate the stability of clustering algorithms. A clustering algorithm is defined as stable when its output (the obtained partition) is minimally affected if its input (the software system) is slightly modified. Stability is generally considered to be a desirable property for a clustering algorithm.

For this experiment, we randomly removed up to 1% of the facts (procedure calls and variable references) from one of our field systems' factbase. Next, we run our clustering algorithm using the reduced factbase and compared the resulting partition to the one obtained from the full factbase. The MoJo values we obtained by running this experiment 100 times can be seen in table 2.

It seems that our clustering algorithm is reasonably stable, since the MoJo distance of the decomposition of the modified software system to that of the original system is less than 10 in 80% of the experiments (the number of modules in this system is 939, which means that in 80% of the experiments the MoJo value computed was less than 1% of the number of modules).

Although this experiment does not absolutely determine the stability of this particular clustering algorithm,[5] it demonstrates that MoJo can be used to see if a clustering algorithm is reasonably stable.

[4] The clustering algorithm has not been published yet, but some of the ideas used in it can be found in [12] and [13].

[5] Further experiments could include aadding dependencies to the software system, as well as adding or removing source files.

MoJo distance	0	1	2	3	4	5	6	7	8	9	10-19	> 20
Frequency	14	11	6	9	9	7	5	9	3	7	11	9

Table 2. Frequencies of obtained MoJo values.

4.3 Clustering quality metric

A metric like MoJo can also be used in order to evaluate the effectiveness of clustering algorithms. For this to happen, we need to have an "authoritative" clustering A of a software system, i.e. a clustering that has been approved by someone knowledgeable about the system, such as the system designers or seasoned developers. We can then evaluate how close a clustering algorithm M comes to reproducing the authoritative clustering A by computing its distance to the clustering B obtained from the algorithm.

We propose the following quality metric for clustering algorithms (n is the number of objects to be clustered):

$$Q(M) = (1 - \frac{MoJo(A, B)}{n}) \times 100\%$$

It is trivial to show that $MoJo(A, B) < n$ (any partition of n objects can be transformed into any other partition of the same set of objects by performing n *move* operations). Therefore, $Q(M)$ will always have a value between 0% and 100% (100% is only achieved if $A = B$, i.e. the clustering algorithm produces the same partition as the system designers).

After running our clustering algorithm (let us denote it by M_0) on one of our test systems we obtained:

$$Q(M_0) = (1 - \frac{345}{939}) \times 100\% = 63.2\%$$

This result was encouraging, since it can be a great help for reverse engineering projects if an automatic algorithm can correctly cluster a large portion of the software system at hand. As a means of comparison, we computed the quality of our random clustering generator. The best value obtained after creating 10000 partitions was 23.4%, while the average value was approximately 12.8%.

This quality metric can also be used to evaluate future versions of a clustering algorithm in order to decide if they constitute an improvement or not.

4.4 Parameter tuning

Many approaches presented in the software clustering literature are parameterized, i.e. they require appropriate choices for the values of certain parameters in order to maximize their effectiveness. MoJo can aid the process of choosing these values by means of the aforementioned quality metric.

For this experiment we used Bunch[6], a clustering tool that employs genetic algorithms[7] (among others), and requires choice of value for its parameters (default values are provided but they may not be the optimal ones). The following table presents experiments with various values for certain parameters[6] and the corresponding value of the quality metric (one of our field systems was used). From these results, one can deduce that a larger population size and crossover probability, together with a smaller number of generations seem to be the appropriate choices.

Number of generations	Population size	Crossover probability	Q(Bunch)
500	50	0.5	48.35%
500	50	1	49.41%
500	100	0.5	47.39%
500	100	1	52.08%
1000	50	0.5	43.88%
1000	50	1	40.04%
1000	100	0.5	41.32%
1000	100	1	43.45%

Table 3. Quality metric values for various combinations of values for 3 parameters.

It is easy to see that such experimentation can help the user of a parameterized tool maximize its performance.

5 Conclusions and Future Work

We have presented MoJo, a metric that can help the process of comparing and evaluating software clustering approaches. We have also presented a heuristic algorithm for the effective calculation of its value.

We believe that the experiments presented in this paper have sufficiently demonstrated that MoJo can be used efficiently in a variety of reverse engineering scenaria providing an objective and fast measure of similarity between software clusterings.

Our intent is to employ this metric in order to research the stability of known software clustering algorithms, and present a comparative study of our findings. MoJo will also be used to study and improve the clustering algorithm we are developing.

[6] The number of generations, the population size, and the crossover probability are common parameters of genetic algorithms.

Acknowledgements

The work presented here has been supported by the Consortium for Software Engineering Research (funded by IBM Canada and by the Natural Science and Engineeering Research Council of Canada), as well as by CITO (Communications and Information Technology Ontario).

The authors would also like to thank Spiros Mancoridis for his help concerning Bunch.

References

[1] N. Anquetil and T. Lethbridge. File clustering using naming conventions for legacy systems. In *Proceedings of CASCON 1997*, pages 184–195, Nov. 1997.

[2] S. C. Choi and W. Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, pages 66–71, Jan. 1990.

[3] P. Finnigan et al. "The Software Bookshelf". *IBM Systems Journal*, 1997.

[4] D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, 11(8):749–757, Aug. 1985.

[5] A. Jain and R. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.

[6] S. Mancoridis, B. Mitchell, et al. Using automatic clustering to produce high-level system organizations of source code. In *IEEE proceedings of the 1998 International Workshop on Program Comprehension*. IEEE Computer Society Press, 1998.

[7] M. Mitchell. *"An Introduction to Genetic Algorithms"*. The MIT Press, Cambridge, Massachusetts, 1997.

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

[9] J. M. Neighbors. Finding reusable software components in large systems. In *Proceedings of the Third Working Conference on Reverse Engineering*, pages 2–10. IEEE Computer Society Press, Nov. 1996.

[10] V. V. Raghavan and C. T. Yu. A comparison of the stability characteristics of some graph theoretic clustering methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3, 1981.

[11] R. W. Schwanke and M. A. Platoff. Cross references are features. In *Second International Workshop on Software Configuration Management*, pages 86–95. ACM Press, 1989.

[12] V. Tzerpos and R. C. Holt. A hybrid process for recovering software architecture. In *CASCON 1996*, pages 1–6, Nov. 1996.

[13] V. Tzerpos and R. C. Holt. The orphan adoption problem in architecture maintenance. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 76–82, Oct. 1997.