Lossless Comparison of Nested Software Decompositions

Mark Shtern and Vassilios Tzerpos
York University
Toronto, Ontario, Canada
{mark,bil}@cse.yorku.ca

Abstract

Reverse engineering legacy software systems often involves the employment of clustering algorithms that automatically decompose a software system into subsystems. The decompositions created by existing software clustering algorithms are often nested, i.e. subsystems may contain other finer-grained subsystems as well as system resources, such as source files. It is rather surprising then, that almost all existing methods for decomposition comparison assume flat decompositions, i.e. subsystems only contain system resources.

In this paper, we introduce UpMoJo, a novel comparison method for software decompositions that can be applied to both nested and flat decompositions. The benefits of utilizing this method are presented in both analytical and experimental fashion. We also compare UpMoJo to the END framework, the only other existing method for nested decomposition comparison.

1 Introduction

It is difficult to imagine a world without computers nowadays. Many organizations, such as banks, airports, or even smaller companies rely on software systems that must be continuously in operation in order to function properly. As software systems attempt to improve constantly to meet market requirements, they become more and more complicated.

Many of the software solutions in operation today are integrations of multiple software systems from different software houses. This brings additional complexity to the task of understanding such a software system. Design documents that explain the relations between the various components are necessary. However, creating such a document is a challenging task. Commercial reasons may dictate that part of the architecture is not even available.

Moreover, the market often demands new software features in a short time. One of the consequences of this trend is that software systems often have poor documentation. In many cases, design documents are never updated. At the same time, software developers often move to other projects or even different companies. Conversely, software developers are often using code components that were not developed in-house.

Finally, research on code cloning indicates that the exchange of source code through the indiscriminate use of copy/paste facilities is a common practice between software developers. As a result, developers often do not know exactly what is going on in the code they are working with.

These factors create situations where software developers can not predict the system's behaviour. Maintenance activities, such as fixing bugs or developing new features, require a lot of effort from developers and testers. A large amount of important business software is in operation today that developers are simply afraid to change. Organizations are often faced with the dilemma of either switching to a new software product, or reengineering the existing one. Risk assessment processes often indicate the latter approach as the most viable one.

Retrieving design information from the source of a software system is an important problem that affects all stages of the life cycle of a software system. Usually, complicated systems consist of different subsystems that can help divide such a system into logical components. The natural decomposition of a software system is usually presented as a nested decomposition [3]. Many different methodologies and approaches that attempt to cre-

ate such decompositions automatically have been presented in the literature[1, 2, 3, 5, 7, 10]. Most of them produce nested decompositions.

Evaluating the effectiveness of such software clustering approaches is a challenging issue. Comparing results produced by different tools is a complicated problem. A number of approaches that attempt to tackle this problem have been presented [4, 6, 9]. However, all of them assume a flat decomposition. The END framework [8] allows one to reuse a technique developed for flat decompositions in order to compare nested ones, but it requires a weighting vector that may not be available.

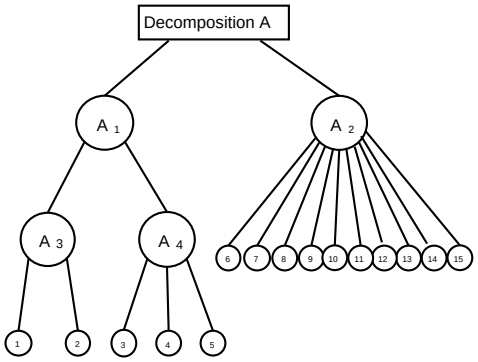
In this paper, we present a new approach to the comparison of nested decompositions. This approach, called UpMoJo, is a generalization of the MoJo distance measure for flat decompositions [9]. UpMoJo adds an Up operation to the existing Move and Join operations of MoJo which allows for differences in the hierarchical structure of two different decompositions of the same system to be reflected in the distance measured.

The structure of the rest of this paper is as follows. Section 2 presents the issues that arise when nested decompositions are flattened in order to be compared using existing methods. Section 3 presents the differences between UpMoJo and the other existing method for the comparison of nested decompositions, the END framework. The UpMoJo distance measure is introduced in Section 4. Experiments that showcase the usefulness of this measure are presented in Section 5. Finally, Section 6 concludes the paper.

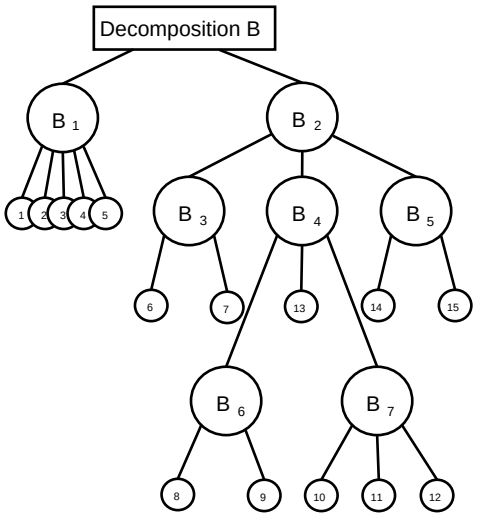
2 Flat vs. nested decompositions

Most of the decomposition comparison methods presented in the literature have been developed in order to compare flat decompositions. Since clustering algorithms commonly create nested decompositions, various methods have been devised in order to evaluate clustering results using these methods. The most common approach is to convert the nested decompositions into flat ones before applying the comparison method. However, this approach has limitations.

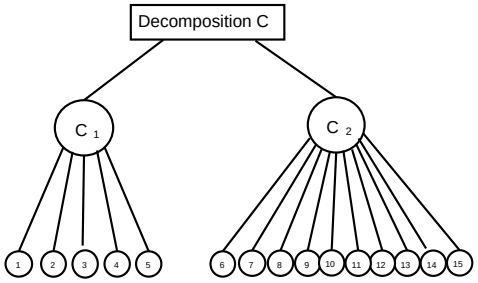
Converting a nested decomposition to a flat one is commonly done in two different ways depending on whether a compact or a detailed flat decomposition is required:



(a) A nested decomposition of a software system

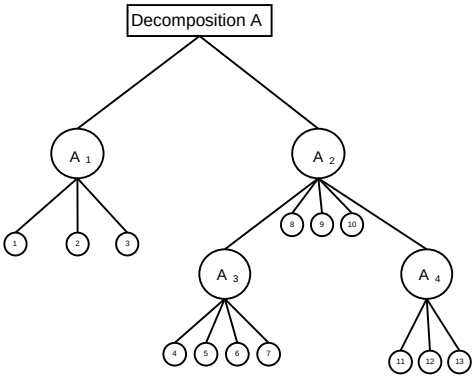


(b) Another nested decomposition of the same software system

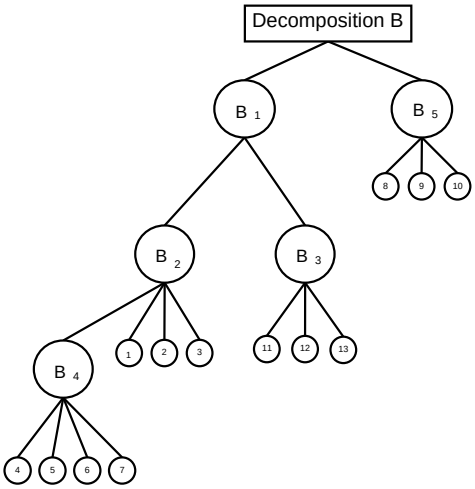


(c) The compact flat form of both decompositions (a) and (b)

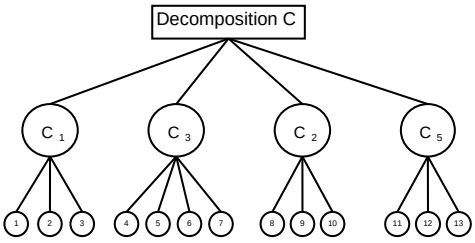
Figure 1. Limitation of compact flat decompositions.



(a) A nested decomposition of a software system



(b) Another nested decomposition of the same software system



(c) The detailed flat form of both decompositions (a) and (b)

Figure 2. Limitation of detailed flat decompositions.

1. Converting a nested decomposition to a compact flat one. Each object is assigned to its ancestor that is closest to the root of the containment tree. The flat decomposition obtained contains only the top-level clusters of the original one.

2. Converting a nested decomposition to a detailed flat one. Each cluster and its sub-tree is assigned directly to the root of the containment tree. The decomposition obtained contains any cluster that contained at least one object in the original decomposition.

Figure 1 presents three decompositions of the same software system. Clearly, decompositions (a) and (b) have different hierarchical structures.

If we convert the decompositions in Figure 1 to compact flat form, they will both become equivalent to decomposition (c). However, the original decompositions were quite different. A significant amount of information has been lost. Figure 2 presents an example where detailed transformation creates a similar problem.

These examples illustrate clearly that converting nested decompositions to flat ones removes significant information that could impact the evaluation process. The UpMoJo comparison method presented in this paper utilizes the whole hierarchy resulting in lossless comparison.

3 The END Framework

The Evaluation of Nested Decompositions (END) framework [8] allows a reverse engineer to compare nested decompositions of large software systems without having to lose information by transforming them to flat ones first. The END framework is able to compare two nested decompositions without any information loss by converting them into vectors of flat decompositions and applying existing comparison methods to selected elements of these vectors. The overall similarity between the two nested decompositions is computed based on the similarity vector that was generated from comparing the decomposition vectors.

The END framework does not specify the function that will be used to convert the similarity vector to a number. While this makes the framework more flexible, it also increases the responsibility of the user. In practice, it is often difficult to differentiate between different weigthing functions as

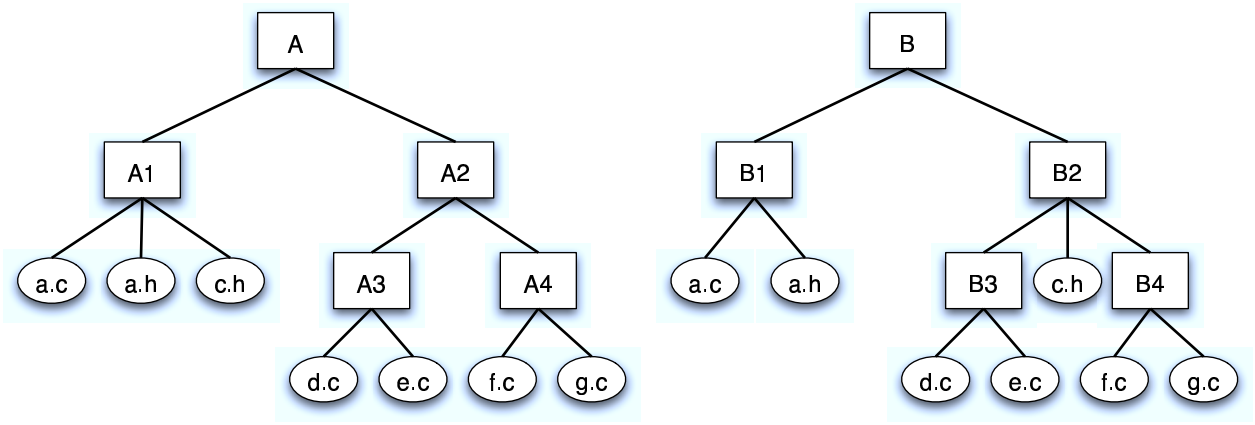


Figure 3. Decompositions `A` and `B`

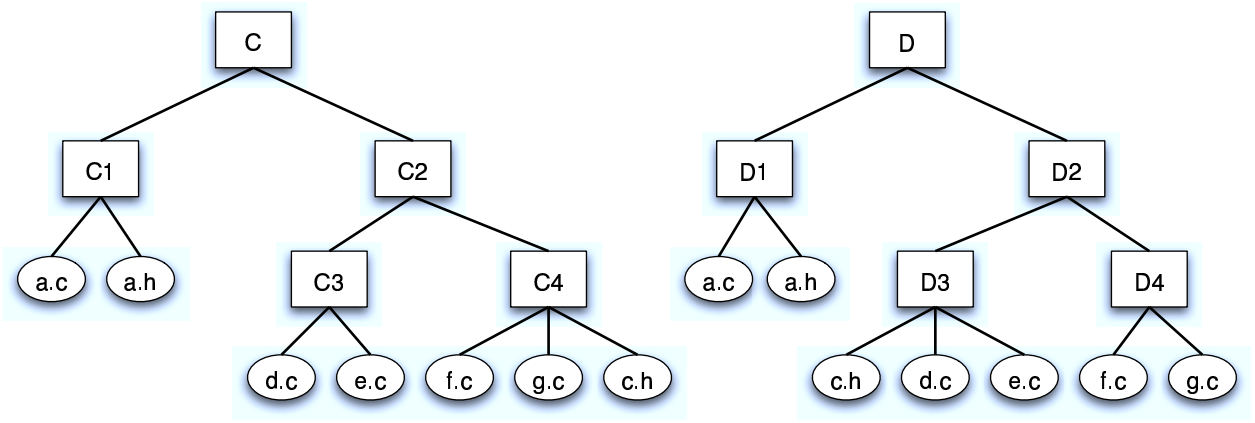


Figure 4. Decompositions `C` and `D`

the framework does not provide any guidelines as to which functions are best suited in which situations. This can result in different users getting different results from END while comparing the same nested decompositions. The UpMoJo measure presented in this paper addresses this problem by providing a comparison method that is solely dependent on the two decompositions.

Moreover, one of the properties of the END framework is that a misplaced object closer to the root of the containment tree results in a larger penalty than if the misplaced object were deeper in the hierarchy. Consider the four decompositions `A`, `B`, `C`, and `D` shown in Figures 3 and 4. The only difference between them is the position of object `c.h`. The END framework would determine that the difference between `A` and `B` is larger than that of `C` and `D` (the similarity vector in the first case is $(1, 1)$, while in the second it is $(0, 1)$). The UpMoJo measure eliminates this discrepancy by penalizing equally for misplaced objects regardless of their position in the containment tree.

Finally, the END framework does not penalize for some trivial hierarchical mistakes. For example, the two nested decompositions shown in Figure 5 are equal according to END. UpMoJo does differentiate between the two structures by assigning a distance larger than 0.

4 The UpMoJo algorithm

This section presents the UpMoJo algorithm through the use of a running example. Let us assume that a software clustering algorithm has produced the nested decomposition shown in Figure 6.

In order to evaluate whether the clustering al-

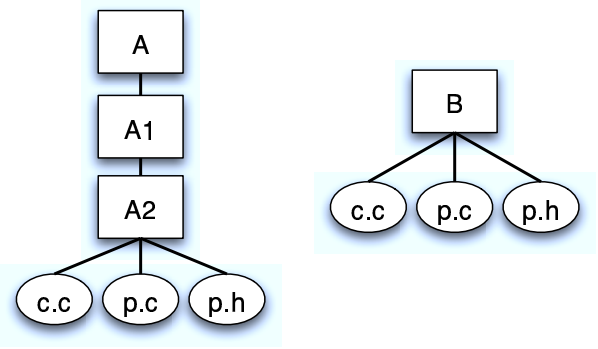


Figure 5. An example of a hierarchical mistake ignored by END.

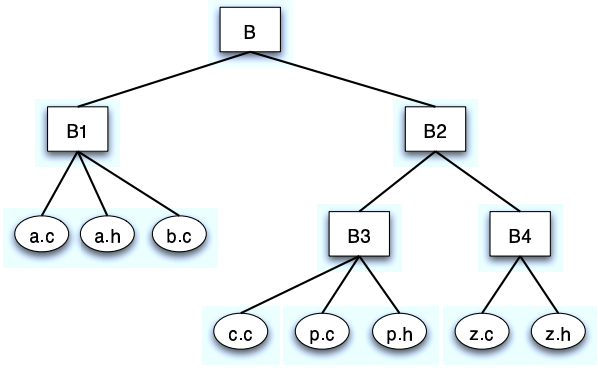


Figure 7. The containment tree of the authoritative decomposition B.

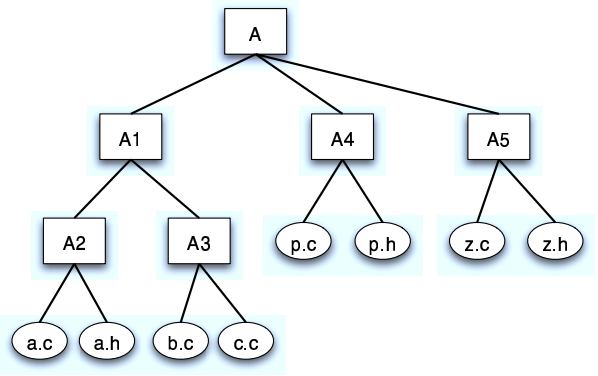


Figure 6. The containment tree of an automatic decomposition A.

gorithm did a good job, we can compare decomposition A to one created by system experts. Let us assume that decomposition B shown in Figure 7 is the authoritative one for our example system.

The UpMoJo distance between the two nested decompositions is defined as the number of operations one needs to perform in order to transform the containment tree of decomposition A to the containment tree of decomposition B. The more operations this transformation requires, the more different the two decompositions are. The rest of this section describes how these operations are counted by our algorithm.

In the following, the *level* of a node n in the tree is defined as the distance from the root to node n, e.g. subsystem A2 in decomposition A is at level 2. The root of a containment tree is at level 0.

We also refer to the system elements being clustered, i.e. the eight files in our example, as *objects*. As a result, a containment tree contains subsystems and objects. All leaf nodes in the tree are objects. All non-leaf nodes are subsystems.

The algorithm that calculates UpMoJo distance is as follows:

Find the set S of objects in level 1 in the authoritative nested decomposition. For each element o of S that is at a higher level than 1 in the automatic decomposition, transform the automatic decomposition by moving o to level 1. For each level that o has to move, increase the UpMoJo distance by 1. Objects that follow the same path move together, i.e. the UpMoJo distance is increased only for one of them. Each move of a set of objects to a lower level is called an Up operation and is explained in more detail in Section 4.1.

In our example, set S is empty, so no Up operations will take place. An example of the above process will be given for the algorithm's next iteration.

Next, we flatten the two decompositions to level 1, i.e. remove subsystems in levels other than 1 (this process does not modify the decompositions but rather creates copies of them). The two flat decompositions in our example are shown in Figures 8 and 9.

We now employ the MoJo distance measure for flat decompositions. The MoJo distance between two flat decompositions of the same set of objects is the minimum number of Move and Join operations one needs to perform in order to transform one flat decomposition into the other. The two types of operations are defined as follows:

- **Mo**ve: Remove an object from a subsystem

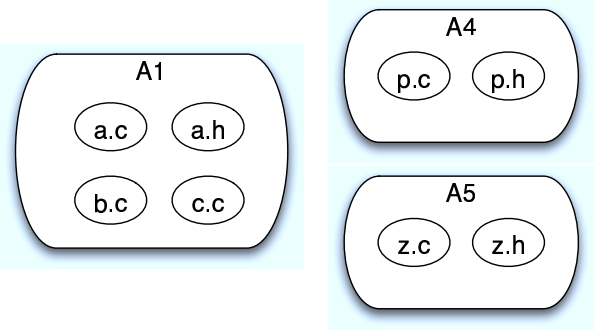


Figure 8. The flat decomposition of nested decomposition A.

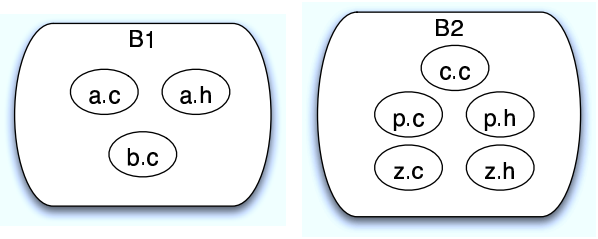


Figure 9. The flat decomposition of nested decomposition B.

and put it in a different subsystem. This includes removing an object from a subsystem and putting it in a new subsystem by itself.

- **Jo**in: Merge two subsystems into one.

In our example, we can transform flat decomposition A to flat decomposition B by joining subsystems A4 and A5 into a new subsystem called A45 and moving c.c to A45 as well.

Having utilized MoJo to determine what are the necessary Move and Join operations for the flattened decompositions, we return to the original decomposition A and transform it in the way suggested by the MoJo distance measure, i.e. join subsystems A4 and A5, and move object c.c. This will result in the containment tree for decomposition A shown in Figure 10 (we denote it by A' to distinguish it from the original version). Subsystem A6 is created so that c.c is at the same level as before. The current total of the UpMoJo distance in our example is now 2 (1 Move and 1 Join operation).

Notice that decompositions A' and B contain the same number of top-level subsystems, and each of

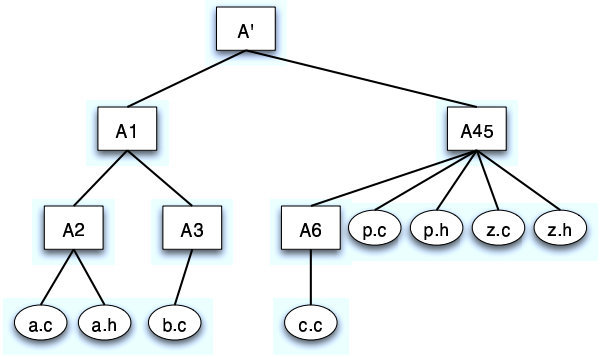


Figure 10. Decomposition A after the transformation indicated by MoJo.

these subsystems transitively contain the same objects. This means that the process of transforming A to B can now continue recursively for each subsystem until A is transformed exactly into B.

The final value of the UpMoJo distance between decompositions A and B is the total number of Up, Move, and Join operations performed during the transformation process.

In our example, the next iteration will try to transform subsystem A1 into subsystem B1. Set S will contain objects a.c, a.h, and b.c (they are in level 1 with respect to subsystem B1). All three will need to move up, since they are initially in level 2 with respect to subsystem A1. However, since two of them reside initially in the same subsystem, only two Up operations are required. This will increase the current total for the UpMoJo distance to 4. It should be easy to see that the flat decompositions will be exactly the same, so nothing further will be required for this subtree.

Finally, for subsystem B2, set S will be empty. The flat decompositions are shown in Figures 11 and 12. Objects directly under the root of the tree are considered to be in a separate subsystem of cardinality 1 in the flat decomposition.

Three further Move operations are required to transform the flat decomposition of A45 to the flat decomposition of B2. Objects p.c, p.h and z.h need to be moved. By performing these operations on the containment tree of decomposition A', we arrive to the containment tree shown in Figure 13, which is identical to the one of decomposition B (the names of the subsystems are immaterial).

As a result, the final value of UpMoJo distance will be 7.

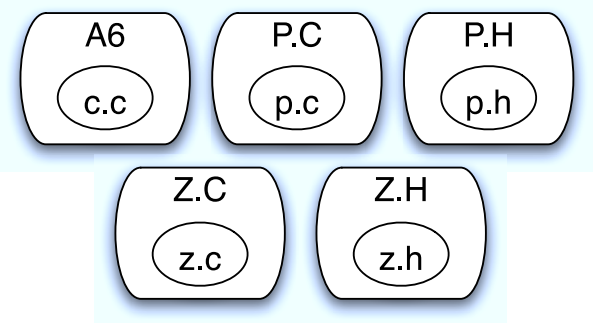


Figure 11. The flat decomposition of subsystem A45**.**

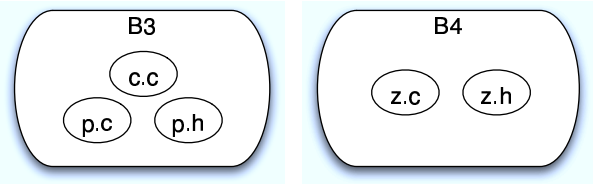


Figure 12. The flat decomposition of subsystem B2**.**

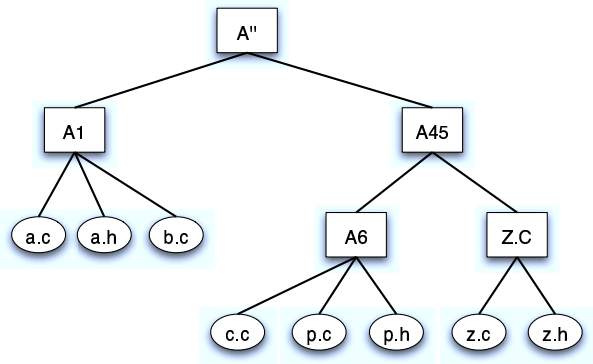


Figure 13. The final containment tree for decomposition A**.**

4.1 UpMoJo Discussion

There are three properties of the UpMoJo algorithm that warrant further discussion:

1. The way the Up operation works.
2. The apparent lack of a Down operation.
3. The fact that UpMoJo does not attempt to compute the minimum number of Up, Move, and Join operations.

The Up operation is necessary when two different decompositions of the same software system have placed a given object at a different hierarchical level. The definition of an Up operation is as follows:

Up: Move an object or a set of objects that initially reside in the same subsystem S to the subsystem that directly contains S.

The intriguing property of the Up operation is that one is allowed to move a set of objects with one operation. Figure 14 shows an example of such an operation. The intuition behind this lies with the fact that these objects have already been placed together by the software clustering algorithm which implies that they are related. The only problem is that they are not at the same hierarchical level as in the authoritative decomposition. It seems unfair that the algorithm be penalized for each object individually, since the most important property, the fact that these objects are related, was discovered.

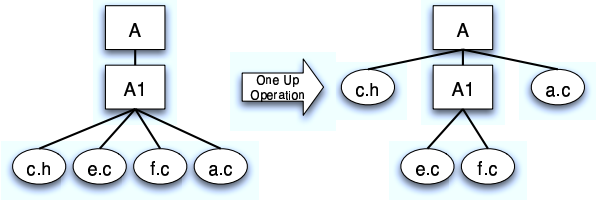


Figure 14. Example of 1 Up operation

The second interesting property of UpMoJo is that there is no Down operation. At first sight, this seems to be strange since the possibility exists that an object will be placed higher in the automatic decomposition than in the authoritative one. However, a downward movement is accomplished by the Move and Join operations. Introducing a Down operation would penalize twice for the same discrepancy between the two decompositions. The following example indicates how this works:

Consider the two decompositions shown in Figure 15. Suppose that we are transforming A to B. It would appear that a Down operation is required. The UpMoJo algorithm will accomplish the same effect implicitly as shown below.

The first step of UpMoJo is to perform any necessary Up operations. It is easy to see that no such operations are required in this example. Next, the flat decompositions for level 1 are calculated.

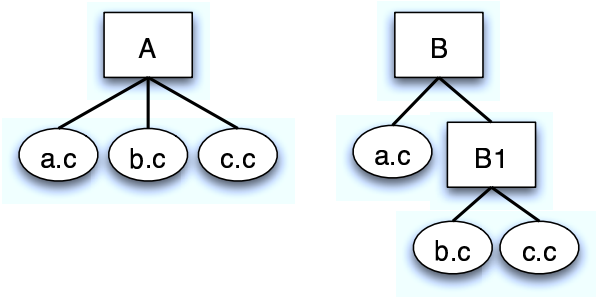


Figure 15. An example of an implicit Down operation

These are shown in Figures 16 and 17 (as explained earlier, objects directly under the root of the tree, `a.c`, `b.c`, and `c.c` in this case, are considered to reside in separate subsystems of cardinality 1 in the flat decomposition). MoJo would indicate one Join operation. This will mean that objects `b.c` and `c.c` are now one level lower in the hierarchy.

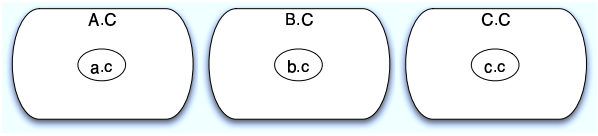


Figure 16. The flat decomposition of A.

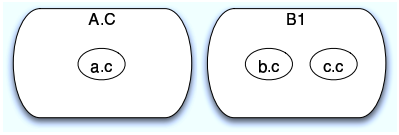


Figure 17. The flat decomposition of B.

Finally, the UpMoJo algorithm does not compute the minimum number of Up, Move and Join operations required to transform one nested decomposition to another. In order to justify this decision, we define a new metric called MinUpMoJo that does compute the minimum number of operations needed to transform the containment tree of decomposition A to the containment tree of decomposition B.

In order to indicate why UpMoJo is a more appropriate measure, we will use the nested decompositions in Figures 18, 19, and 20. Table 4.1 presents results from applying both UpMoJo and MinUpMoJo to these decompositions.

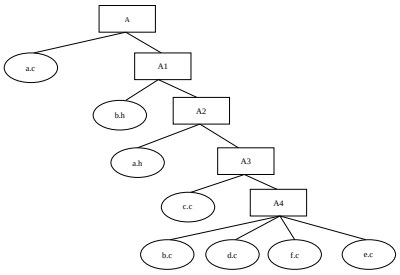


Figure 18. A - Nested Decomposition .

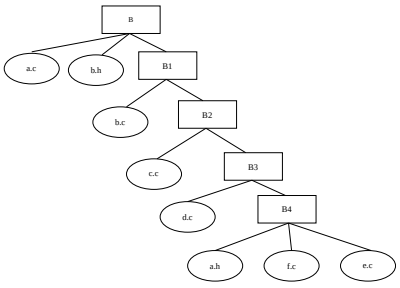


Figure 19. B - Nested Decomposition .

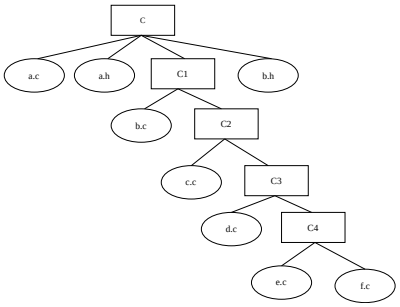


Figure 20. C - Nested Decomposition .

	(A,C)	(B,C)
MinUpMoJo	4	4
UpMoJo	7	4

Table 1. MinUpMoJo and UpMoJo results

MinUpMoJo has determined that both A and B are equally different from C. This result is misleading since B is clearly closer to C than A. The optimization behaviour of MinUpMoJo has masked the differences between A and C. On the other hand, the UpMoJo method ranked the nested decompositions correctly and confirmed that B is closer to C.

5 Experiments

The advantages of comparing nested software decompositions in a lossless fashion should be apparent by now. When one flattens a decomposition, important information that could distinguish two decompositions may be lost. While this is certainly true in theory, it is not apparent whether it makes a significant difference in practice.

The experiments presented in this section attempt to answer the following questions:

1. Does UpMoJo produce different results in practice than a comparison method for flat decompositions such as MoJo?

2. What are the practical differences between UpMoJo and the END framework using a trivial weighting function, such as equal weight for all values in the similarity vector?

Before we present the process we followed in order to answer these questions, we discuss the following example. Suppose we want to compare three nested decompositions A, B, and C. By applying the MoJo method we obtain the results in Table 1.

MoJo(A,B)	MoJo(B,C)	MoJo(A,C)
1	2	3

Table 2. MoJo results

When we compare the same nested decompositions using UpMoJo we obtain the results in Table 2.

UpMoJo(A,B)	UpMoJo(B,C)	UpMoJo(A,C)
5	2	8

Table 3. UpMoJo results

According to MoJo, A and B are the two most similar decompositions. However, according to UpMoJo, this is incorrect. This is a significant difference because it affects the way the two algorithms rank the three decompositions. In other words, we would not consider it a significant difference if UpMoJo produced larger distance values than MoJo as long as the two algorithms ranked all decompositions in the same order of similarity.

A generalization of this idea will be the congruity metric we will use in order to compare UpMoJo to both MoJo and END. Assume we have N pairs of nested decompositions of the same software system. We can apply both MoJo and UpMoJo to each pair and obtain two different values m_i and u_i. We arrange the pairs of decompositions in such a way so that for $1 \leq i \leq N-1$ we have $m_i \leq m_{i+1}$. The value of the congruity metric will be the number of distinct values of i for which $u_i > u_{i+1}$. Values for this metric range from 0 to $N-1$. A value of 0 means that both comparison methods rank all pairs in exactly the same order, while a value of $N-1$ probably indicates that we are comparing a distance measure to a similarity measure. Values significantly removed from both 0 and $N-1$ indicate important differences between the two comparison methods. This is the result we were hoping for in our experiments.

In order to perform the evaluation described above, we generated 100 random nested decompositions containing 346 objects with an average height of 4 and calculated MoJo and UpMoJo values for all pairs. We repeated this experiment 10 times to ensure that our results are not based on an unlikely set of decompositions. In all experiments, the value of the congruity metric ranged from 35 to 44. This clearly indicated that there are significant differences between MoJo and UpMoJo. Using MoJo to compare nested decompositions runs the risk of producing results that do not reflect the inherent differences between the decompositions.

We performed the same comparison between END and UpMoJo. In the case of END, we used MoJo as the plugin comparison method, and applied a weighting vector that contained equal weights for all values of the similarity vector. The same experiment setup as before was used (100 random decompositions, 10 repeats). The values of the comparison congruity metric ranged from 25 to 42, indicating again a significant difference between END and UpMoJo. Section 3 mentions the main differences between END and UpMoJo. It is up to the reverse engineer to decide which set of features is more appropriate for their project.

Table 4 presents further experimental results that confirm the above findings (they also add the expected result that END is significantly different from MoJo). The table presents values for the congruity metric for an experiment setup of 100 random nested decompositions and various combina-

Comparison methods	D = 6 C = 4	D = 8 C= 4	D = 8 C = 20	D = 8 C = 60	D = 12 C = 60
MoJo and UpMoJo	37	42	50	43	46
MoJo and END	25	31	42	46	41
END and UpMoJo	32	41	36	41	50

Table 4. Metric values for several experimental setups.

tions of values for two generation parameters: D is the maximum possible depth of the generated nested decomposition, and C is the maximum possible children for any subsystem in the generated decomposition.

These experimental results indicate clearly that the three comparison methods provide significantly different results.

6 Conclusions

This paper presented a novel comparison method for nested software decompositions called UpMoJo. UpMoJo takes differences in the hierarchical structure of the nested decomposition into account and allows for comparison that does not lose any information due to conversion to flat decompositions. At the same time, UpMoJo is able to operate without the need for additional input, such as the weighting vector required by END. Our experiments indicate that UpMoJo does indeed produce significantly different results than both MoJo and END.

References

[1] P. Andritsos and V. Tzerpos. Software clustering based on information loss minimization. In *Proceedings of the Tenth Working Conference on Reverse Engineering*, pages 334–344, Nov. 2003.

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

[3] 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.

[4] R. Koschke and T. Eisenbarth. A framework for experimental evaluation of clustering techniques. In *Proceedings of the Eighth International Workshop on Program Comprehension*, pages 201–210, June 2000.

[5] 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 the International Conference on Software Maintenance*. IEEE Computer Society Press, 1999.

[6] B. S. Mitchell and S. Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *Proceedings of the International Conference on Software Maintenance*, pages 744–753, Nov. 2001.

[7] H. A. Müller and J. S. Uhl. Composing subsystem structures using (k,2)-partite graphs. In *Conference on Software Maintenance*, pages 12–19, Nov. 1990.

[8] M. Shtern and V. Tzerpos. A framework for the comparison of nested software decompositions. In *Proceedings of the Eleventh Working Conference on Reverse Engineering*, pages 284–292, Nov. 2004.

[9] V. Tzerpos and R. C. Holt. MoJo: A distance metric for software clusterings. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 187–193, Oct. 1999.

[10] V. Tzerpos and R. C. Holt. ACDC: An algorithm for comprehension-driven clustering. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 258–267, Nov. 2000.