ACDC : An Algorithm for Comprehension-Driven Clustering

Vassilios Tzerpos

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

R. C. Holt

University of Waterloo
Waterloo, Ontario, CANADA
holt@plg.uwaterloo.ca

Abstract

The software clustering literature contains many different approaches that attempt to automatically decompose software systems. These approaches commonly utilize criteria or measures based on principles such as high cohesion and low coupling, information hiding etc.

In this paper, we present an algorithm that subscribes to a philosophy targeted towards program comprehension and based on subsystem patterns. We discuss the algorithm's implementation and describe experiments that demonstrate its usefulness.

1 Introduction

A common approach that researchers in various disciplines use in order to deal with large data sets is to develop a taxonomy, i.e. create categories of objects that exhibit similar features or properties. Such categories (commonly referred to as *clusters*) can be discovered through a variety of techniques that have been proposed in the literature. Research on the effectiveness and behaviour of these techniques has given rise to the field of *cluster analysis*.

Software engineering is a relatively new discipline compared to some of the other research areas that utilize cluster analysis techniques.[1] However, in recent years the size of the typical industrial-strength software system has increased considerably. Understanding such large pieces of software is a difficult task. For that reason, several techniques that attempt to decompose a software system into meaningful subsystems have been developed. They are collectively referred to as *software clustering* techniques.

Software clustering techniques presented in the literature generally employ certain criteria in order to decompose a software system. Such criteria include low coupling and high cohesion, interface minimization, shared neighbours etc. However, it appears that in an effort to maximize the performance or accuracy of their algorithms, many researchers have lost sight of the fact that the primary concern of software clustering is comprehension. Our first priority should not be to ensure that the clusters produced by our approach satisfy certain criteria, but that they can be effective in helping someone understand the software system at hand.

For this reason, we believe that an effective software clustering algorithm should produce output that can be easily understood, either by proposing clusters that follow familiar patterns, or by naming these clusters intelligently. Also, the size of the obtained clusters can be an important factor. A cluster containing more than one hundred objects is not very useful, even though it might exhibit a high degree of cohesion. We should strive to keep the size of the clusters at a more manageable level (up to 20 objects or so). Coupled with effective visualization techniques, such an approach can produce clusterings that are essential for the comprehension process of a software system.

In this paper, we present a software clustering algorithm that subscribes to this philosophy. It discovers clusters that follow patterns that are commonly observed in decompositions of large software systems that were prepared manually by their system architects. It also assigns automatically meaningful names to the clusters and tends to avoid creating clusters of very large size. The name of our algorithm is ACDC (for an **A**lgorithm for **C**omprehension-**D**riven **C**lustering).

The structure of the rest of this paper is as follows: Section 2 discusses some background on software clustering techniques that have been presented in the literature. Section 3 presents our point of view on clustering for program comprehension. Common subsystem patterns are listed in section 4. Section 5 presents our

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

algorithm and its features. Section 6 describes several experiments we conducted with large industrial systems to demonstrate the usefulness of our algorithm. Finally, section 7 concludes the paper.

2 Background

There are two common ways to approach the problem of identifying clusters in a large software system:

1. *Knowledge-based* approach. Using such an approach, one attempts to understand what different pieces of source code do by utilizing reverse engineering techniques and pre-existing domain knowledge. Program modules that implement similar or complementary functionality can then be grouped together, e.g. procedures implementing mathematical functions can be assumed to be part of the same library.

2. *Structure-based* approach. In this case, the decomposition of a software system is determined by looking at syntactic interactions (such as "call" or "fetch") between entities (such as procedures or variables). The problem of clustering a software system can be thought of as the partitioning of the vertex set of a graph, where the nodes are defined as procedures or variables, and the edges as relations between these entities.

Although knowledge-based approaches have been shown to work well with small systems, they do not perform as effectively when dealing with large ones. Various reasons, such as the size of the knowledge base becoming prohibitively large, the lack of problem domain specific semantics, and knowledge spreading in the source code contribute to this phenomenon [Nei96]. The majority of software clustering researchers has concentrated on structure-based techniques.

In one of the early works in software clustering, L. A. Belady and C. J. Evangelisti [BE81] recognized the need to automatically cluster a software system in order to reduce its complexity. They also presented a first approach to doing this for a specific system. In addition, they provided a measure for the complexity[2] of a system after it has been clustered. Their approach, however, only works with a specific kind of system, and they did not validate their complexity measure. A point of interest is that they didn't extract information from the source code, but rather from the system's documentation.

Subsequent to this work, Hutchens and Basili [HB85] performed clustering based on *data bindings*. A data binding was defined as an interaction between two procedures based on the location of variables that are within the static scope of both procedures. They defined different kinds of data bindings; from simplistic and easy to compute to sophisticated and hard to compute. On the basis of the data bindings, a hierarchy is constructed from which a partition could be derived.

An interesting feature of their paper is that they compared their structures with the developer's mental model with satisfactory results. They also raised the important issue of *stability*; when the system changes slightly, the effect on the algorithm's output should be minimal as well. Finally, they recognized that it might be necessary to disregard certain information in order to get a clearer view of the structure of a software system.

One of the most active researchers in the area of software clustering in the early 1990s was Robert W. Schwanke. His papers [SAP89, SP89, Sch91] and his tool (called ARCH) addressed the problem of automatic clustering in an innovative way. Although his approaches were not tested against a large software system, they showed considerable promise. One of his main contributions was that he added to the "classic" low-coupling and high-cohesion heuristics by introducing the "shared neighbors" technique [SP89] in order to capture patterns that appear commonly in software systems. Also, his "maverick analysis" [Sch91] enabled him to refine a partition by identifying components that happened to belong to the wrong subsystem, and placing them in the correct one.

Choi and Scacchi [CS90] presented an approach to finding subsystem hierarchies based on resource exchanges between modules. The complexity of their algorithm is $O(n^2)$, which is better than Schwanke's $O(n^3)$, but still probably too high for large systems. It appears to perform well on small examples, but its ability to scale up is questionable.

Hausi Müller has also been involved in the automatic clustering problem [MU90, MOTU93]. His approaches tend to be semi-automatic, meaning that they are meant to help a designer perform clustering on a software system. He introduces the important principles of *small interfaces* (the number of elements of a subsystem that interface with other subsystems should be small compared to the total number of elements in the subsystem) and of *few interfaces* (a given

[2] Complexity here refers to how difficult it is to understand a system after it has been clustered in a specific way.

subsystem should interface only with a small number of the other subsystems).

Arun Lakhotia introduced a unified framework for expressing software clustering techniques [Lak97]. Realizing that the techniques in the software clustering literature have been presented using different terminology and symbols, he proposed a framework consisting of a consistent set of terminology, notation, and symbols, that can be used to describe the input, output, and processing performed by these techniques. Several existing techniques were reformulated to conform to this framework.

An interesting alternative approach was presented by Nicolas Anquetil and Timothy Lethbridge [AL97]. Instead of looking at structural information, such as procedure calls or data references, they only looked at the names of the resources of the system. Their experiments produced promising results, but their approach has the obvious drawback that it relies on the developers' consistency with the naming of their resources.

Ivan Bowman and R.C. Holt introduced the term *ownership architecture* in [BH99]. They argued that the organization of system developers into teams can help understand a software system, and that such a structure is often congruent to the system's concrete architecture. An extensive case study involving the Linux operating system was presented to corroborate their conjecture.

Finally, various researchers have recently started looking at techniques used in other disciplines in order to come up with a better solution to the automatic clustering problem.

Theo Wiggerts [Wig97] presented a survey of techniques used by the cluster analysis community and attempted to reuse them for system remodularization. His future plans included clustering a software system in a "more or less object-oriented" way.

Several researchers attempted to use concept analysis in order to identify subsystems [LS97, vDK99]. Their experiments demonstrate that concept analysis could be helpful in certain reverse engineering scenaria, such as object identification.

Spiros Mancoridis [MM+98] treated clustering as an optimization problem and employed genetic algorithms in order to overcome the local optima problem of "hill-climbing" algorithms, which are commonly used in clustering problems. His experiments demonstrate encouraging results and fast performance.

By examining the literature on software clustering one can draw interesting observations. First, most researchers seem to agree on structural-based criteria and naming conventions as being the most promising approaches. However, there exists a variety of different interactions between modules that are used as the basis to decide which resources depend on which. Isolating the interactions that are appropriate for the software clustering problem, and determining the properties that make them so, is a problem that needs more study.

Another observation is that none of the approaches has been tested extensively against large software systems[3]. This omission becomes more interesting when one considers that these approaches were developed with such systems specifically in mind. It is not clear whether these approaches scale up to large systems.

Also, validation of an approach against more than one system is required. Many researchers present results that demonstrate that their algorithm performs very well for a given software system. It would be interesting to see how the algorithm performs on a number of systems, since an algorithm can be specifically tuned to perform well on a particular system.

The issue of performance is also important. Most graph partitioning problems are shown to be NP-complete [GJ79] or NP-hard. The approaches presented above, however, are mostly heuristic approaches that attempt to reduce this complexity to polynomial upper bounds. What kind of complexity is acceptable for large systems remains to be determined.

Finally, the usefulness of these approaches is undoubtedly hard to measure. Empirical studies would have to be conducted in order to deem any of these techniques effective in a real-life environment.

In the next section, we present an algorithm that exhibits certain features that we believe would make it a good candidate for usage with industrial-strength systems.

3 Clustering for comprehension

Most of the algorithms presented in the software clustering literature identify clusters by utilizing certain intuitive criteria, such as the maximization of cohesion, the minimization of coupling, or some combination of the two. Such criteria can definitely produce meaningful clusterings, a fact that is showcased by the success such approaches have had with a variety of software systems.

However, we believe that an excessive amount of effort has been put into fine tuning these techniques.

[3] A large system refers to an industrial system with a size of several hundred thousand lines of code.

In an effort to improve the accuracy or performance of their algorithms, researchers have lost sight of the primary goal of software clustering, comprehension. Rather than trying to discover a clustering that exhibits slightly larger cohesion values, one should strive to cluster the software system in a way that will aid the process of understanding it.

It is a well-known fact that any clustering problem does not have a single answer [Eve93]. Depending on the point of view of the clustering procedure, more than one clusterings can serve as the answer. Hence, it is more important to attempt to produce a clustering that will be helpful to the developers trying to understand the software system, rather than to try to find the clustering that maximizes the value of some metric.

We believe that a software clustering algorithm should have certain features that will ensure that its output will be helpful to someone attempting to understand the software system at hand. These features are:

- *Effective cluster naming.* Providing meaningful names for the obtained clusters is an issue that has not attracted much attention (Schwanke acknowledged the importance of this problem and presented a semi-automatic solution in [SP89]). We believe that it is important that the high level view of a software system contains subsystems with familiar names, rather than names such as Subsystem01, SS02 etc. Effective naming allows people associated with the software system to understand the obtained decomposition faster. This means that they can refine it more easily, and start benefiting from it more readily.

- *Bounded cluster cardinality.* A partition of a system's resources where one cluster contains the overwhelming majority of the resources while the remaining clusters contain one or two resources each, can hardly be useful, even though it might exhibit low coupling. On the other hand, clusters containing a limited number of objects (up to 20 or so) are more manageable and easier to understand. An algorithm targeted towards program comprehension should attempt to create decompositions that have this feature. This should not be done however at the expense of the system's inherent structure. The algorithm should simply decompose any large clusters further, producing nested clusters of more than one level if necessary.

- *Pattern-driven approach.* Humans can understand something more easily if it is presented in familiar terms or patterns. Our experience indicates that certain patterns emerge time and again when humans create manual decompositions of software systems they are knowledgeable of. This suggests that if a decomposition contains these patterns, it would be easier to understand.

  For this reason, we believe that an algorithm that identifies these patterns, and creates a system decomposition based on them can be successful in helping the comprehension process of a software system.

The next section presents a list of subsystem patterns that are commonly observed in manual decompositions of large industrial systems.

4 Subsystem patterns

From city planning to object-oriented programming [GHJV95], it is well-known that patterns are followed in many a creative process. In a software reengineering context, a pattern might refer to a process that alleviates certain legacy software problems [DRN99]. When it comes to software clustering, patterns refer to familiar subsystem structures that frequently appear in manual decompositions of large industrial software systems.

The remainder of this section presents several subsystem patterns. It is important to note that these patterns are found in large systems. They might not necessarily apply to smaller systems (a bare minimum of 100 source files is probably necessary for some of these patterns to emerge).

Following is a list of possible subsystem patterns:

1. *Source file pattern.* Depending on the programming language used, a source file may contain the definition of one or more procedures/functions, as well as the declaration of several variables. If a clustering algorithm is attempting to cluster resources at the procedure/variable level, then the set of procedures and variables contained in the same source file can be grouped together into one cluster.

2. *Directory structure pattern.* Important clustering information is sometimes encoded in the directory structure of the source code. In other words, directories may correspond to subsystems under certain circumstances. However, one has to be careful as this is not always the case. For example, many systems contain directories that are

collections of header files. Such a cluster might not be useful from a comprehension point of view.

3. *Body-header pattern.* Many programming languages are designed in a way that results in a procedure being split between two different files, e.g. a .c and a .h file in C. This pattern lumps such files together into a cluster (usually containing only two files). This is a special type of subsystem where low cardinality is acceptable, if not desirable. If presented to the developers effectively, it can reduce the complexity of a system's structure significantly while the amount of information conveyed is intact.

4. *Leaf collection pattern.* A pattern commonly observed in software systems is a set of files that are not connected to each other, i.e. not dependant on each other, but they serve similar purposes, such as a set of drivers for various peripheral devices. These files are usually leaves in the system's graph, hence the name (this pattern is similar to the "shared neighbors" approach presented by Schwanke [SP89]).

5. *Support library pattern.* Software systems commonly contain a number of procedures that are accessed by the majority of its subsystems (such procedures are also referred to as "omnipresent nodes" [MOTU93]). This pattern groups such resources together into one subsystem, something that usually makes discovering the structure of the rest of the system an easier task.

6. *Central dispatcher pattern.* This pattern is the dual of the support library pattern. Large systems commonly contain a small number of resources with a large out-degree, i.e. they depend on a large number of other resources. A common example is a procedure (commonly called a driver) that calls may other procedures in order to execute certain parts of the system in sequence.

   The proliferation of edges originating from such a resource might obscure other patterns in the system's structure. For this reason, one initially disregards such resources and their outgoing edges, and later reconsiders them in conjunction with already formed subsystems.

7. *Subgraph dominator pattern.* This pattern looks for a particular type of subgraph in the system's graph $G = (V, E)$. This subgraph must contain a node n_0 (called the "dominator node") and a set of nodes $N = n_i, i : 1..m$ (called the "dominated set") that have the following properties:

   (a) There exists a path from n_0 to every n_i.

   (b) For any node $v \in V$ such that there exists a path P from v to any $n_i \in N$, we have either $n_0 \in P$ or $v \in N$.

   In simpler terms, one must go through node n_0 in order to get to any node $n_i \in N$.

   This pattern can be extended so that instead of having a "dominator node" we have a "dominating set of nodes". However, this increases the complexity of the task of finding such subgraphs significantly, while it is not clear that the obtained clusters would be easier to understand. For this reason, we will not consider the extended version here.

This list of patterns is by no means complete. Depending on the development philosophy adopted by different teams, or the specific nature of certain software systems, more patterns may be added.

An important point that should not be overlooked is that different patterns may suggest different clusterings for the same software system. Therefore, an algorithm must decide which patterns are appropriate, as well as how to combine information obtained from different patterns in a meaningful and effective way.

In the next section, we will discuss how ACDC accomplishes this task. We will also present further details about the way our algorithm functions.

5 The ACDC algorithm

ACDC performs the task of clustering a software system's resources in two stages. In the first one, it creates a skeleton of the final decomposition by identifying subsystems using a pattern-driven approach. Depending on the pattern used, the subsystems are given appropriate names (by convention, subsystem names contain the suffix ".ss"). In the second stage, ACDC completes the decomposition by using an extended version of a technique known as Orphan Adoption [TH97]. We will look at both stages in more detail.

5.1 Stage 1: Skeleton construction

As noted in section 4, there is a variety of patterns that a pattern-driven clustering approach can attempt to identify. ACDC considers many of those patterns and performs the following steps in order of precedence:

1. *Source file clusters.* It is typically beneficial for the comprehension of a software system to cluster system resources in a way that follows the source file pattern. Therefore, ACDC clusters resources that reside in the same file together, and considers the source file as being an atomic entity for the following steps (the name of the source file will also be the name of the cluster, an exception to the rule that all subsystem names end in ".ss").

2. *Body-header conglomeration.* In terms of the conceptual structure of a software system, the interface and the implementation of a software module are parts of the same entity. For this reason, ACDC consolidates them into one subsystem. The name of this subsystem will be the common name of the two files with the extension ".ss", e.g. *foo.c* and *foo.h* will be grouped together in a subsystem called "foo.ss".

3. *Leaf collection and support library identification.* In this step ACDC identifies collections of files that could be candidates for subsystems according to the leaf collection and support library patterns. The support library files are distinguished by the fact that their in-degree is larger than 20.[4] ACDC does not form any subsystems at this point.

4. *Ordered and limited subgraph domination.* This is the main step of our algorithm. To begin, ACDC follows the central dispatcher pattern and disregards any files with an out-degree larger than 20 (see footnote 4). Edges originating at these files are also not taken into account in the following.

   Next, our algorithm goes through all the nodes and examines whether they qualify as the dominator node of a subsystem following the subgraph dominator pattern. The nodes are considered in order of ascending out-degree, i.e. ACDC starts with the node that has the smallest number of outgoing edges and works its way to the node with the largest (ties are resolved arbitrarily).

   The reason for the ascending order is the following: Nodes with smaller out-degree are more likely to induce smaller subgraph dominator pattern instances. By examining them first, when we get to nodes of larger out-degree, there is a better chance that the larger subgraph dominator pattern instances will contain already formed subsystems. This results in smaller cardinality for the final subsystems.

   If a non-empty dominated set is discovered, ACDC creates a subsystem containing both the dominator node and the dominated set. The name of this subsystem is the name of the dominator node plus the suffix ".ss". Nodes in the dominated set are not considered as possible dominator nodes any more, unless the cardinality of the dominated set was larger than 20. In that case, they are not removed from the possible dominator node list, in an effort to further decompose the newly discovered subsystem.

   After all nodes have been examined, ACDC organizes the obtained subsystems, so that the containment hierarchy is a tree (it is possible that the aforementioned process has discovered two subsystems, one of which is a proper subset of the other). It might also remove some subsystems of small cardinality (the contents of a subsystem with a cardinality of 4 or less are moved to the higher level subsystem if that does not increase its cardinality above the threshold of 20).

   Finally, any files that were disregarded earlier are now considered again. If they follow the subgraph dominator pattern (where now the dominated set can contain either files or subsystems), then a new subsystem is formed in the same fashion as above.

5. *Creation of "support.ss".* In this final step of the skeleton construction process, ACDC creates a subsystem called "support.ss". Any files that were identified as candidates for the support library pattern in step 3 are assigned to this subsystem, unless they were already assigned to some subsystem during step 4.

Even though a large proportion of the system's resources is already clustered at this point (see section 6.3), many files are still not assigned to a subsystem, since they did not fit any of the aforementioned patterns. The second stage of our algorithm will attempt to complete the system's structure by assigning all remaining files to one of the existing subsystems.

5.2 Stage 2: Orphan Adoption

Orphan Adoption is a technique that has been developed to deal with the problem of maintaining a system's decomposition as the system evolves. In other

[4] The choice of this particular value (20) is guided by the fact that ACDC attempts to produce clusters of low cardinality (between 5 and 20). As a result, a node with an in- or out-degree larger than 20 should probably be placed in a different cluster than its neighbours.

words, it is an incremental clustering technique. Based on the assumption that the existing structure is well-established, it attempts to place each newly introduced resource (called an orphan) in the subsystem that seems more appropriate. This is usually a subsystem that has a larger amount of connectivity to the orphan than any other subsystem (more details on Orphan Adoption can be found in [TH97]).

In our case, the skeleton decomposition that we constructed in the previous stage serves as the existing structure, while the non-clustered files are the orphans. The only difference to the usual application of Orphan Adoption is that the number of orphans will probably be somewhat larger.

ACDC employs this technique in order to assign all files to some subsystem. A small extension is introduced when several files that were earlier identified as candidates for the leaf collection pattern cannot be assigned to any subsystem with confidence, i.e. more than one subsystems are possible parents.

If that is the case, ACDC creates a new subsystem that will contain these files. The name of this subsystem will be "leaf.ss", unless these files happen to reside in the same directory. In that case, the directory's name plus the suffix ".ss" will be used. Alternatively, an analysis for common substrings (as in [AL97]) can be performed in order to derive a more appropriate name for this subsystem.

5.3 Algorithm properties

It is interesting to note that the ACDC algorithm does have the features that were presented in section 3 as essential for a software clustering algorithm:

- Any time ACDC creates a new subsystem, it provides a name that is either intuitive or will be familiar to the people associated with the software system in question.

- The cardinality of the obtained clusters is indeed bounded. Step 4 of the skeleton construction process (that creates the majority of the subsystems) attempts to create subsystem containing no more than 20 files. Larger subsystems are decomposed further if possible. The Orphan Adoption process may add some extra contents into some subsystems, but that number is usually insignificant.

- The pattern-driven nature of the algorithm is obvious. Several different patterns are considered, which should produce an output suitable for a program comprehension project.

A feature of the obtained decomposition that may not be immediately obvious is that it is nested and unbalanced. This usually happens because instances of the subgraph dominator pattern may contain other instances (figure 1 presents a rather trivial example). We consider this to be a positive feature of our algorithm since it reflects the real-life situation of many software systems (certain parts are usually more complicated than others).

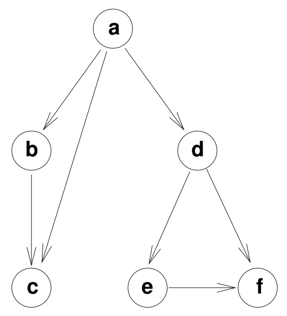

Figure 1: An example of nested subgraph dominator pattern instances. Node d will be examined first, and a subsystem containing nodes d, e, and f will be formed. When node a is examined, another subsystem will be formed containing nodes a, b, c, and the previously formed subsystem.

Also, our algorithm makes rather limited use of the directory structure pattern. This is mainly due to our experience with several large industrial systems, where the directory structure did not seem to reflect the structure of the system, or was even non-existent. However, this does not mean that the directory structure pattern is of no use. If it is considered an appropriate pattern for a given software system, its integration with our algorithm is easy (an extra step can be added between steps 2 and 3).

Finally, our algorithm utilizes a number of seemingly arbitrary values, such as the cardinality bounds of 5 and 20. These values are not fixed, and can actually be viewed as parameters. However, we do not believe that choosing other reasonable values for these parameters would affect the effectiveness of our algorithm significantly. That is why we chose to present the algorithm using fixed values.

In order to gauge the usefulness of our algorithm we conducted several experiments. The next section presents our results.

6 Algorithm validation

No software clustering algorithm can claim to be successful, unless it has been tested on real-life systems. For this reason, we implemented ACDC as described in section 5, and tried it on two large systems of comparable size, but of different development philosophy:

1. **TOBEY**. This is a proprietary industrial system that is under continuous development. It serves as the optimizing back end for a number of IBM compiler products. The version we worked with was comprised of 939 source files and approximately 250,000 lines of code.

2. **Linux**. We experimented with version 2.0.27a of this free operating system that is probably the most famous open-source system. This version had 955 source files and approximately 750,000 lines of code.

In the following, we describe the experiments we conducted and present the obtained results. All the experiments were run on a Sun Enterprise 450/4000 machine running Solaris 2.5.1.

6.1 Performance

The execution speed of a clustering algorithm is not the most decisive factor for its evaluation. However, it is still desirable that the algorithm completes in a reasonable amount of time.

ACDC appears to perform well in that aspect. On the specific hardware mentioned above, it required 54 seconds in order to cluster TOBEY. Linux, a somewhat larger system, required 84 seconds. Experience with other comparable systems shows that ACDC always finishes in a reasonable amount of time.

6.2 Stability

A software clustering algorithm is defined to be stable, if its output is not significantly affected when the input software system is modified. It is recognized that it is desirable for an algorithm to be stable.

In order to measure ACDC's stability, we employed the measure presented in [TH00]. This measure randomly perturbs the algorithm's input and ascertains the effect on the algorithm's output. If that effect is less than a predefined threshold, the algorithm is deemed to have behaved in a stable fashion. By repeating this a large number of times, one can get a good feel of the algorithm's stability.

The results of our experiment showed that ACDC behaved in a stable fashion in 81.3% of the repetitions when using TOBEY as input. In the case of Linux, the corresponding number was 69.4%. Both of these numbers are definitely satisfactory, and indicate that ACDC is a stable algorithm. This makes it a good candidate for reverse engineering projects working with systems that are under development, since it predicts that clusterings of successive "snapshots" of the system will not be significantly different.

6.3 Skeleton size

An important condition that has to be satisfied in order to use the Orphan Adoption procedure is that a significant structure already exists. In our case, that means that the skeleton that is constructed in the first stage of our algorithm must contain a significant proportion of the system's resources.

Experiments showed that the TOBEY skeleton contained 604 of the 939 files of the system (a percentage of 64.3), while for Linux that number was 488 out of 955 (51.1%). Both of these values indicate that a well-formed structure is already in place before the Orphan Adoption process starts adding to it.

6.4 Quality

The goal of this experiment was to show that ACDC's focus on program comprehension does not have a negative effect on the decompositions it produces, i.e. they still come close to the software system's inherent structure. For this purpose, we employed the quality measure presented in [TH99], which measures how close a given clustering comes to an authoritative decomposition produced by people knowledgeable about the system.

For our experiment, we obtained the authoritative decomposition for TOBEY from its developers. The Linux one was obtained from the Software Bookshelf of the Linux Kernel [Bre98]. The quality measure values that we obtained were 64.2% for TOBEY and 55.7% for LINUX. These values indicate that ACDC produces meaningful results as well as ones suitable for program understanding. The quality measure for TOBEY especially, is one of the higher ones an automatic clustering algorithm can hope to achieve.

6.5 Observations

Table 1 presents a summary of the results of the experiments presented in this section.

	Performance	Stability	Skeleton size	Quality
TOBEY	54 sec	81.3%	604/939 (64.3%)	64.2%
Linux	84 sec	69.4%	488/955 (51.1%)	55.7%

Table 1: Summary of results.

An interesting observation that is apparent from the results of our experiment is that ACDC performs better with TOBEY rather than with Linux. The quality measure values, as well as the stability results are strong indicators of that fact. The size of the TOBEY skeleton is also considerably larger than the Linux one.

This phenomenon can probably be attributed to the nature of the two systems. TOBEY is a system developed by a relatively small-sized developing team that follows strict company regulations and a formal design. It is only natural that its structure can be more easily discovered in an automatic way than Linux, a system developed by many people residing in various geographical locations and following different development philosophies.

Overall however, the experiments we presented in this section suggest that ACDC can be an effective software clustering algorithm. The quality of the clusterings it produces, along with its comprehension-driven approach should make it a good candidate for reverse engineering projects.

7 Conclusions

We believe that a software clustering algorithm targeted towards program comprehension should regulate its output so that it aids the understanding of the given software system. This can be achieved by coming up with aptly-named subsystems of familiar structure and reasonable size.

Accordingly, we presented an algorithm that operates in a pattern-driven fashion by attempting to discover subsystems that resemble ones commonly observed in manually decomposed subsystems. We described the implementation of our algorithm and explained why we think it could help the process of understanding a software system. Finally, we presented experiments that showed the usefulness of our approach.

Acknowledgements

The work presented here has been supported by CSER (Consortium for Software Engineering Research), IBM Canada, the Natural Science and Engineering Research Council of Canada, and CITO.

References

[AL97] Nicolas Anquetil and Timothy Lethbridge. File clustering using naming conventions for legacy systems. In *Proceedings of CASCON 1997*, pages 184–195, November 1997.

[BE81] L. A. Belady and C. J. Evangelisti. System partitioning and its measure. *Journal of Systems and Software*, 2:23–29, 1981.

[BH99] Ivan T. Bowman and R. C. Holt. Reconstructing ownership architectures to help understand software systems. In *Proceedings of the Seventh International Workshop on Program Comprehension*, May 1999.

[Bre98] Neil Brewster. "The Software Bookshelf of the Linux Kernel". *http://www.turing.toronto.edu/~brewste/bkshelf/linux/*, 1998.

[CS90] Song C. Choi and Walt Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, pages 66–71, January 1990.

[DRN99] Stephane Ducasse, Tamar Richner, and Robb Nebbe. Type-check elimination: Two object-oriented reengineering patterns. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 157–166, October 1999.

[Eve93] Brian S. Everitt. *Cluster Analysis*. John Wiley & Sons, 1993.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability*. W. H. Freeman and co., 1979.

[HB85] David H. Hutchens and Victor R. Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, 11(8):749–757, August 1985.

[Lak97] Arun Lakhotia. A unified framework for software subsystem classification techniques. *Journal of Systems and Software*, pages 211–231, March 1997.

[LS97] Christian Lindig and Gregor Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 19th International Conference on Software Engineering*, pages 349–359, May 1997.

[MM+98] Spiros Mancoridis, B.S. 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.

[MOTU93] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5:181–204, December 1993.

[MU90] Hausi A. Müller and James S. Uhl. Composing subsystem structures using (k,2)-partite graphs. In *Conference on Software Maintenance*, pages 12–19, November 1990.

[Nei96] James 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, November 1996.

[SAP89] Robert W. Schwanke, R.Z. Altucher, and Michael A. Platoff. Discovering, visualizing, and controlling software structure. In *International Workshop on Software Specification and Design*, pages 147–150. IEEE Computer Society Press, 1989.

[Sch91] Robert W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proceedings of the 13th International Conference on Software Engineering*, pages 83–92, May 1991.

[SP89] Robert W. Schwanke and Michael A. Platoff. Cross references are features. In *Second International Workshop on Software Configuration Management*, pages 86–95. ACM Press, 1989.

[TH97] Vassilios 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, October 1997.

[TH99] Vassilios 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, October 1999.

[TH00] Vassilios Tzerpos and R. C. Holt. On the stability of software clustering algorithms. In *Proceedings of the Eighth International Workshop on Program Comprehension*, June 2000.

[vDK99] Arie van Deursen and Tobias Kuipers. Identifuing objects using cluster and concept analysis. In *Proceedings of the 21th International Conference on Software Engineering*, pages 246–255, May 1999.

[Wig97] Theo A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 33–43. IEEE Computer Society Press, October 1997.