Assessing Software Archives with Evolutionary Clusters*

Adam Vanya [1,†], Lennart Hofland [2], Steven Klusener [1], Piërre van de Laar [3], Hans van Vliet [1]
[1] Computer Science Department, VU University Amsterdam, The Netherlands
[2] Philips Medical Systems, Best, The Netherlands
[3] Embedded Systems Institute, Eindhoven, The Netherlands

Abstract

The way in which a system's software archive is partitioned influences the evolvability of that system. The partition of a software archive is mostly assessed by looking at the static (include, call) relations between the parts. Some of the previous work also takes history information into account to assess the partition but they only relate pairs of software entities. In this paper we describe a novel history-based approach to assess the extent in which a certain partition allows its parts to evolve independently. We use the assumption that a set of software entities which co-evolved often in the past are likely to be modified together in the near future as well. Hence, the elements of such a set should in principle belong to the same part. Our approach, therefore, identifies sets of co-evolving software entities, where each set has elements from different parts of the archive. We illustrate our approach with a case study of a large software system that evolved during more than a decade, and has over 7 million lines of code.

1 Introduction

Software degrades, as a consequence of maintenance activities unless effort is invested to keep the structure of the software clean [11]. Even if we carefully design a system such that its parts can be independently developed and maintained, this structure will erode over time. Our systems become legacy systems that are difficult to comprehend and maintain. Software entities may have ended in a branch of a code tree where they do not really belong. Since the structure of the archive is often used to assign development or test tasks, a degraded structure of the archive hampers evolution. Re-partitioning the archive of a legacy software system and assigning separate, independent development life cycles to the identified parts may help mitigate maintenance problems.

Such an improved partition allows one to exploit the benefits of parallel development, like shorter time to market. Since a modification in a given part has a known scope, tests can be applied in a more focused manner and consequently can result in a software product with better quality attributes.

Even if the archive is not yet partitioned, people with a deep knowledge of the system may have a clear idea what the parts of the archive should be. Alternatively, they may know it roughly, meaning that the borders between the parts are somewhat blurred. In the former case the challenge is to assess the partition, while in the latter case experts could be helped to sharpen the borders of the parts. Both problems are addressed by the approach we describe in this article.

Although many other possibilities exist, we tackle these problems by considering the *history* of the system. History information on a software system tells us how, when and why a given software entity was modified. This information typically spans many years and is stored in version management systems, like ClearCase or CVS. We address the following research question in this paper: How can the available history information on a software system help experts divide the archive of a legacy software system into parts which can evolve as independent as possible?

Our approach to deal with the posed research question (1) identifies evolutionary clusters (sets of software entities which changed frequently together in the past), (2) relates the evolutionary clusters to the current partition, (3) indicates structural problems which need to be resolved to arrive to the desired partition. Similar to Tudor Gîrba et al. [6] we use the Yesterday's Weather assumption: if software entities were modified together in the past, they will most likely be modified together in the near future as well. We illustrate our approach with a case study of a large software system containing more than 7 million lines of code, developed over a period of more than a decade.

Previous research either used clustering of pieces of

*This work has been carried out as part of the DARWIN project at Philips Medical Systems under the responsibility of the Embedded Systems Institute. This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.

†Corresponding author. E-mail address: vanya@cs.vu.nl

static information to obtain system-level knowledge (for example the recovery of the architecture), or accumulated history information at the level of individual elements to observe trends and patterns (for example how the features of a class change over time). We combine these approaches: we cluster history information of individual files to assess the system-level partition of a software archive.

In Section 2 we describe how our work relates to the state-of-the-art. Section 3 provides an overview of our approach and introduces the used terminology. In Section 4 we describe our approach step by step. Section 5 details the application of our approach in a case study. In Section 6 we discuss the lessons learned as well as possible directions for future research. In Section 7 we describe our conclusions.

2 Related Work

2.1 History-Related Research

History information about how a software system evolved over time is widely used in the literature, with different purposes. On the one hand, some analyze the history of software entities or their relationships to identify trends and patterns. They do so with the purpose to better understand the system and / or to classify its entities. On the other hand, a significant part of the literature which uses history information identifies modification similarities or couplings among software entities (files, classes, packages).

From the first category, Greevy and Ducasse [8] analyze how the number of features a given class implements changes over time. Their goal is to show how features evolve with respect to their implementations. Lungu and Lanza [12] visualize the evolution of static dependencies, like class inheritance or method invocation between modules, to enrich software exploration. Their tool makes it possible to classify relations based on the underlying patterns of evolution. Others analyze how the ownership of files [7] or how the lines in a file [20] have changed over time. Both works aim at a better understanding of past changes and developer interactions.

As for the second category, Gall et al. [5] uncover similarities or "logical couplings" of classes to pinpoint some of the possible weaknesses of the module structure. Zimmerman et al. [24] use the identified modification similarities among fine-grained software entities (functions, variables) to warn developers for inconsistent changes. Fischer et al. [4] use versioning information of closely related software products to identify the platform of a future product family.

In both these strands, the history information is not clustered. For instance, if the history information is at the level of classes, it is not grouped to obtain knowledge at the package or subsystem level.

2.2 Recovering Software Components

Many papers address the issue of isolating parts of a legacy software system that can evolve independently. Mehta and Heineman [15] for example identify fine-grained components of a legacy system. In order to reach their goal, they execute regression tests related to a given feature.

Reverse engineering subsystems and other higher level entities of a legacy system based on the similarities between identifier names and comments is the goal of Kuhn et al. [10]. They use Latent Semantic Indexing to cluster parts of the legacy system having the same meaning. A similar approach is used by Anquetil et al. [2] but they rely on the names of source files only.

Schwanke [16] uses static information (call relations, returns, common variables) among procedures to calculate their similarities. Based on that, they execute hierarchical, agglomerative clustering to identify the modules of the software system. Wiggerts [22] provides a classification of clustering algorithms and a description of how they can be applied to re-structure a legacy system. Maqbool and Babri [14] review and characterize clustering algorithms which might be used for recovering architectural components. They also assign semantics to the parameters of the clustering algorithms.

Mancorodis et al. [13] identify modules by creating an initial partition of source files and by applying optimization algorithms to decrease the dependencies between the modules and increase dependencies inside the modules. The dependencies considered are static relationships.

Wierda et al. [21] present a case study to recover subsystems of a legacy software by clustering static relations of classes using different versions of the system. They show that using different versions of the system as an input can improve the accuracy of the results.

2.3 Discussion

The literature discussed falls into two disjoint classes:

- Research using history information to characterize entities or their relationships.
- Research aimed at recovering high level software entities, like subsystems or modules, from the source code. These approaches cluster static information (class, procedure relations, naming conventions).

We contribute to both research areas: our work uses history information (check-ins from version management systems) and clusters the information to assess the partition of the software archive. Unlike earlier history-based approaches to assess the partition of an archive, our approach clusters and next relates sets of files instead of only relating pairs of files.

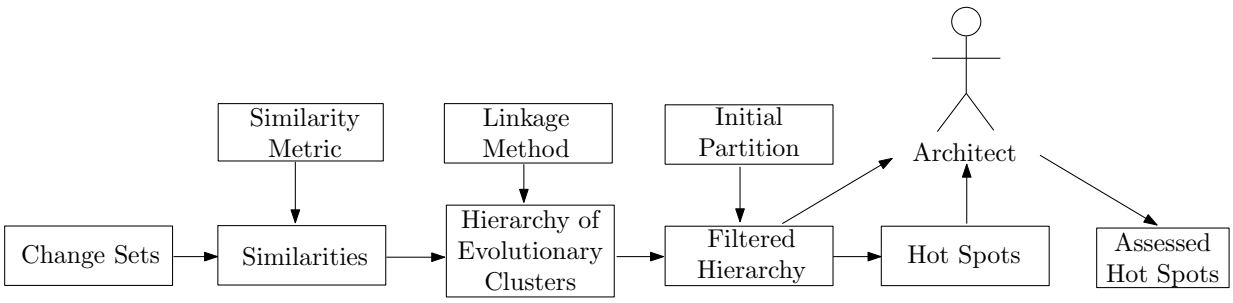

Figure 1. The evolutionary cluster approach

3 Approach Overview

In this section we introduce the terminology used in this article and give an overview of our approach. A more detailed, step-by-step description is the topic of the following section.

We assume that an initial idea of how to partition the software archive into independently modifiable parts is available and we refer to it as the *Initial Partition*. This Initial Partition may come from the documentation, expert knowledge, or the current structure of the archive.

To validate the Initial Partition from the history perspective, our approach may use any type of input capturing which files were modified together in the past because of the same motivation. This motivation can be, for example, the tasks of the developer to modify or to create features. While describing our approach in general we also specify how versioning information residing in Version Management Systems can be used as a specific input for our approach. Versioning information is available most of the time. It can be considered as meta–data of file modifications, like who altered a file, when, how and why. Although advanced Version Management Systems are capable of fine–grained versioning, our approach relies only on the basic functionalities of early, but still widely used, Version Management Systems like CVS.

Eventually, we are interested in real evolution-related issues concerning the Initial Partition, i.e., different parts that were intended to evolve independently, while in the practice they co-evolved. These issues are the result of an assessment process, in which the architects are asked to evaluate a set of *hot spots*. A hot spot indicates that a group of files from one part of the Initial Partition were often changed together with a group from another part. A hot spot does not need to become an issue. For instance, the architect may know and accept that different parts of the Initial Partition co-evolve.

To fully achieve the desired decomposition of the software archive from the evolution perspective, the issues selected by the above process have to be addressed. Our approach, however, stops at assessing the Initial Partition. Describing possible solutions to address an issue is not within the scope of this article.

Figure 1 depicts the steps and sub–steps of our approach. All steps of our approach are further elaborated in the next section. When doing so, we also describe how we designed the steps and what the considered alternatives were.

4 From Change Sets to Evolutionary Clusters

4.1 Change Sets and Their Approximations

The construction of our evolutionary clusters starts with the notion of a *change set*. A change set is a set of modifications that are assumed to have a common motivation. The co-evolution of files can now be measured by their common change sets. If the modifications of files occur in the same change set then those files co-evolved.

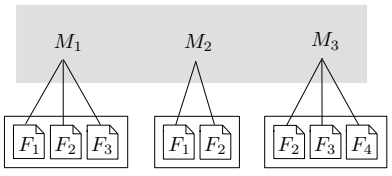

Figure 2. Change Sets

Figure 2 gives an example of how change sets are related to files ($F_i, i \in [1..4]$) and motivations ($M_j, j \in [1..3]$). In the figure, each file symbol denotes a modification of a single file and rectangles denote change sets. The example shows that for instance F_1 and F_2 co-evolved twice, as they were both modified because of M_1 and M_2 but not because of M_3.

Identifying change sets is not obvious, there are various ways to do this. Some change management systems, like Unified Change Management (UCM), provide change sets

directly, see also [3]. In most cases, however, change sets must be approximated from the available data.

In order to construct our approximated change sets, in this paper we group modifications, based on developer and time. We sort all modifications of one developer, based on their time stamp. Consecutive modifications are put in the same change set, unless their difference in time is larger than δ. We assume some given δ, typically a couple of minutes. Figure 3 illustrates the relations between check-ins and change sets.

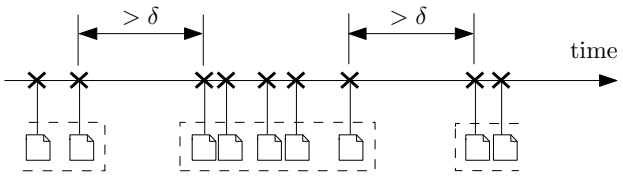

Figure 3. Change Set approximations

This approximation of change sets is similar to the notion of a transaction by Zimmermann et al. [23]. Note that we leave the rationale of each change set implicit, whereas Zimmermann applies a filter based on some rationale constructed from the comments of the modifications.

Approximations introduce false positives and false negatives by their nature. In our case, one developer working on different tasks in parallel may check in files at the same point without a common motivation, which would introduce a false positive. On the other hand, a developer may take a cup of coffee during a sequence of related modifications. This will result in a false negative as the sequence gets interrupted and the modifications will end up in two different change sets.

We do recognize that the way we obtain our change sets influences the construction of the evolutionary clusters. An in-depth discussion of this issue, in particular the dependence on the actual software process of an organization and how a project administration could be used as source of information, is outside the scope of this paper.

4.2 The Similarity Metric

Having constructed the change sets, we can now compute the similarity coefficients for each pair of files. This results in a similarity matrix. Given a pair a file, a similarity coefficient close to 1 reflects that the files often co-evolved, whereas a similarity coefficient close to 0 means that the files hardly co-evolved. The metric we use to identify similarities is the Jaccard coefficient [1]. Given the number times when files were modified together (a_{11}) and separately (a_{10}, a_{01}), the Jaccard similarity coefficient is computed as follows:

$$\frac{a_{11}}{a_{11} + a_{10} + a_{01}}$$

One of the main reasons why the Jaccard similarity coefficient is widely used in the literature and also by us lies in its intuitive semantics. In our case, the coefficient is the probability that two files will co-evolve if one of them gets changed.

4.3 Hierarchy of Evolutionary Clusters

The main contribution of this paper is how evolutionary clusters are constructed from historical information. The Jaccard formula gives us the similarity coefficient between two files, the next step is to aggregate file-level similarity to the similarity between sets of files. Given two sets of files, there are various ways to aggregate the file-level similarity to the level of the sets, also called *linkage methods*. One can use, for example, the minimum of all pairwise coefficients (also known as *single linkage*), the average (*average linkage*) and the maximum (*complete linkage*).

We start with sets containing one file only and we join sets until one set remains. In every step the two sets with the highest linkage value is joined.

This algorithm is known as Agglomerative Hierarchical Cluster Analysis (AHCA) [17, 22]. The resulting hierarchy of clusters is a binary tree, also known as *dendrogram*.

We have chosen the average linkage since the other linkage methods result in a cluster hierarchy being very sensible to even slight modifications to the strength of file similarities. A relative stability of results is needed if we want to base longer term decisions on the identified clusters.

Furthermore, single and complete linkage methods represent two extremes. On the one hand, single linkage method is good at maximizing the cohesion inside clusters, but it tends to leave strongly related clusters separated. On the other hand, the usage of complete linkage results in clusters having low coupling, but also low cohesion. To overcome these problems and have strong similarities inside the clusters and weak similarities between them, we chose the average linkage.

4.4 Evolutionary Clusters

In the resulting dendrogram the nodes are the identified evolutionary clusters. The vertical alignment of the evolutionary clusters in the dendrogram indicates the cohesion level: the nearer an evolutionary cluster is to the root of the dendrogram (the top of Figure 5), the less frequently the contained files co-evolved together.

A set of files may indicate issues with the Initial Partition if there is at least one pair of files in the set which co-evolved frequently and the files belong to different parts of

the Initial Partition. The more such file pairs there are in the set, the more frequently the elements of the set co-evolved and the more parts they belong to, the more relevant the issue might be. Ideally, such border crossing sets should not exist at all. At the minimum they should be known. On the other hand there might also be sets of files which co-evolved frequently inside the same part of the archive.

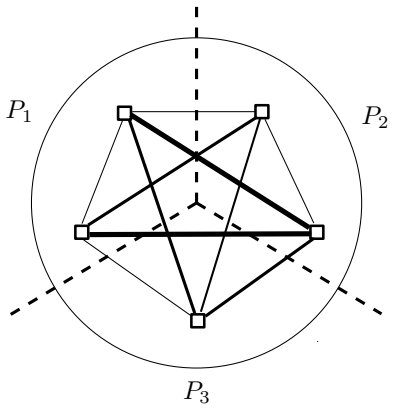

Figure 4. Modifications over the borders of parts

Figure 4 shows a set of co-evolving files from different parts of the Initial Partition. The borders of the parts are indicated by dashed lines, where $P_i, i \in [1..3]$ are the names of the parts. Squares denote files and solid lines represent their modification similarity. The thicker a line between two files is drawn the more similar they are. The circle indicates an evolutionary cluster.

Using the modification similarities computed in Section 4.2, we identify the evolutionary clusters. We are not only interested in files which were always changed together, but also in others which co-evolved less frequently. This makes it possible to perform a finer tuned assessment of the Initial Partition. Frequently, but not always co-evolving files can still hinder independent development.

4.5 The Filtered Hierarchy

Evolutionary clusters containing files from the same part of the Initial Partition do reflect a proper subdivision of the archive. To identify issues they are of less interest, and therefore we call them *trivial*. We may remove these trivial evolutionary clusters from our dendrogram, leading to a pruned one. In the sequel we will focus on non-trivial evolutionary clusters, clusters that contain files from different parts of the Initial Partition.

4.6 Identifying Hot Spots

Even when the Initial Partition is created by people who have a deep knowledge of the software system, it is very well possible that there are modification similarities between the parts that may hamper evolution. As described in sections 4.3 and 4.4, such possible issues are indicated by non-trivial Evolutionary Clusters. In order to assess the Initial Partition the indicators of potential issues should be identified.

By pruning the dendrogram in the previous step, we not only identify Evolutionary Clusters which can be indicators of issues with the Initial Partition, but we also provide a structure for the indicators. The leaves of the tree indicate the hot spots of potential issues and by moving toward the root of the tree (top of Figure 5) we get more contextual information (the scope of the issue), more files, but less cohesion. Also, between leaves there are typically differences in cohesion level. The cohesion level specifies how frequently a set of files co-evolved. By sorting the hot spots in decreasing order of their cohesion level, it is possible to give an ordered list of the hot spots.

The ordered list of hot spots, which is the output of this step, is a suggestion to experts and developers as to which potential evolution-related issues should be addressed first. On the other hand, experts are free to pick any of the hot spots they want to start with. Further, the tree structure has the advantage that more problems can be resolved in a parallel manner.

4.7 Evaluation of Hot Spots

Only potential issues are indicated by the elements of the ordered list, the output of the previous step. Experts may not consider all the strong modification similarities crossing the borders of parts unwanted. One of the reasons is that often when the Initial Partition is created, other requirements are also taken into account, like the execution architecture. In those cases typically a trade-off is made between localizing modifications and other requirements.

When a potential issue is found to be a real risk for the future evolution of the software system, it has to be resolved. Our approach stops at the point when the Initial Partition is assessed. By resolving the issues we can get to the desired partition of the archive from the evolution point of view.

5 Case Study

We have applied our approach on an industrial case to

- show that our approach works in practice
- elaborate how effective our approach is

- generate and answer practical questions related to the application of our approach
- describe an example implementation

5.1 Case Study Environment

The subject of our case study is the software embedded in a complex medical system. The analyzed software system consists of more than 7 million lines of code and more than 34,000 files are involved. The first release of the system was deployed nearly two decades ago and up till now it has been continuously evolving. We performed the case study within the context of the Darwin project. In [18] the Darwin project is further described.

5.2 Our Approach at Work

In this section, for every step of our approach we describe our implementation, what kind of practical issues we faced and how we addressed those issues.

5.2.1 Recovering Change Sets

The used version management system from which we extracted versioning data was ClearCase. Using a ClearCase script, we queried what was checked-in, when and by whom. We extracted check-in related information from the last 6 years of development. The extraction resulted in a script file which we used to transfer check-in information into a table in our SQL database.

At this point we realized that we faced a serious scalability issue. Among the 34,000 files which were checked-in the possible number of relations is more than 1 billion. This is more than what computing environments like Matlab can handle. We recognized, however, that working on the file level would be also too detailed. Fortunately, our approach can work on higher abstractions than files.

To overcome the scalability issue and remain focused, instead of files we considered the directories representing the next level of abstraction on the file hierarchy. This way, we consider the check-in of a file as the check-in of a directory. Massaging the SQL table accordingly resulted in check-ins of more than 500 directories. These directories are the implementations of *Building Blocks* [19], [9].

To identify change sets, we implemented a sliding window algorithm as an SQL stored procedure and applied the stored procedure on the previously extracted check-in meta data. We developed the stored procedure such that the identified change sets are output into a new SQL table.

The analysis of the identified change sets revealed that there were very huge Change Sets, containing even more than 1000 check-ins. During the discussions with developers it turned out that those huge Change Sets are the side effects of merging activities. In order to get rid of this kind of noise we discarded all Change Sets containing more than 100 check-ins. The threshold of 100 was advised by the developers.

5.2.2 Similarities and the Cluster Hierarchy

In-line with step described in Section 4.2 we developed another SQL stored procedure to identify modification similarities among Building Blocks. For that we used the formula from Section 4.2. The result is a table of similarities.

To execute the step detailed in Section 4.3 we transfered the similarity table into the ”R” statistical package. In ”R” we executed the AHCA on the similarity table. The resulting dendrogram, before the reduction, is shown in Figure 5.

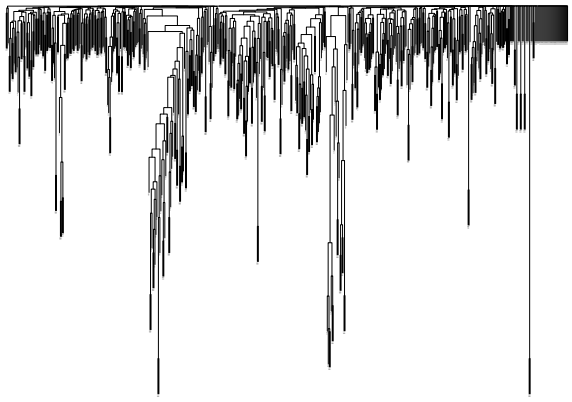

Figure 5. The dendrogram on real data

5.2.3 Reduction and Hot Spot Identification

For the Initial Partition we took the subsystem decomposition of the archive as point of departure. Identifying subsystems was easy, because the Building Blocks of the software system are organized into a hierarchical structure and this structure reflected the subsystem decomposition.

To implement the step described in Section 4.5, first we manually reduced the dendrogram such that it contained non-trivial evolutionary clusters only. By looking at the leaves (hot spots) of the reduced dendrogram we identified the top 10 most interesting evolutionary clusters which may hinder independent evolution of the subsystems. The identified top 10 clusters are the first 10 elements of the ordered list described in Section 4.6, that is, the 10 non-trivial evolutionary clusters having the highest cohesion.

5.2.4 Evaluation of Hot Spots

We gave the list of the top 10 hot spots to the architects and asked if the identified Evolutionary Clusters did indicate real obstacles to the evolution of the subsystems. We also asked whether the indicated issues were known, or whether our approach taught the architects about new problems. We refer to this classification of hot spots with the following symbols:

- $\times$: non-issues (false positives)
- $\sim$: agreed-upon but already known issues
- ! : agreed-upon yet unknown issues

Using this annotation our top 10 list is presented in Table 1. The architects did not find major false negatives.

Table 1. Top 10 Hot Spots

Rank	1	2	3	4	5	6	7	8	9	10
Type	!	$\sim$	!	!	$\sim$	!	$\sim$	$\sim$	$\times$	$\sim$

All the agreed-upon problems which were unknown before indicated surprising relationships between subsystems. In one of those cases our approach identified co–evolving Building Blocks from different subsystems which were designed to evolve independently. The software of these subsystems have been designed to co-evolve with the hardware they are associated with, but not to co-evolve with each other. The result was surprising also, because the contained Building Blocks were created relatively recently.

Except one, all top 10 potential problems were found to point out evolutionary problems. The exception refers to a set containing two Building Blocks. From those two, one was recently removed from the archive and consequently the set is not an indicator of an evolutionary problem any more.

5.3 The interesting cases

In this subsection we further analyze the four cases which taught the architects about yet unknown evolution related issues. From now on we will refer to the related non-trivial evolutionary clusters as interesting clusters. First, we start with indicating at which cohesion level we found the interesting clusters, see Figure 6. Case x refers to the evolutionary cluster with rank x in the top 10 list.

As can be seen even the interesting cluster with the highest cohesion has a rather low cohesion value (0.14). This means that whenever a Building Block is modified in that cluster the probability that another one in the same cluster needs to be modified is at least 0.14 on average.

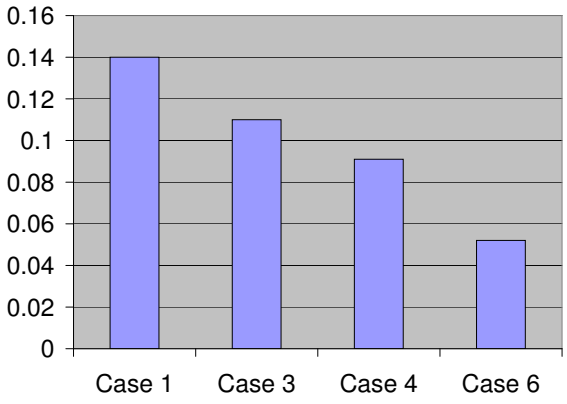

Figure 6. Cohesion level

The reason why this happened is twofold. First, most of the change sets concern one subsystem only (indicate stat). On the top of that, the Jaccard metric produces low values, by its very nature, even if files were modified together relatively frequently. For example, suppose files f_1 and f_2 both changed 4 times. Once they changed together. Then the Jaccard parameters a_{11}, a_{10} and a_{01} are 1, 3 and 3 respectively. So the Jaccard similarity is 0.14 while the probability that f_1 and f_2 are changed together is 25%.

The benefit of applying a symmetric similarity metric is the applicability of the clustering algorithm. Even though we have to work with low similarity values, we are still able to point out relevant evolution related issues.

We continue the analysis of the four cases by showing properties of the modifications when the directions are also taken into account.

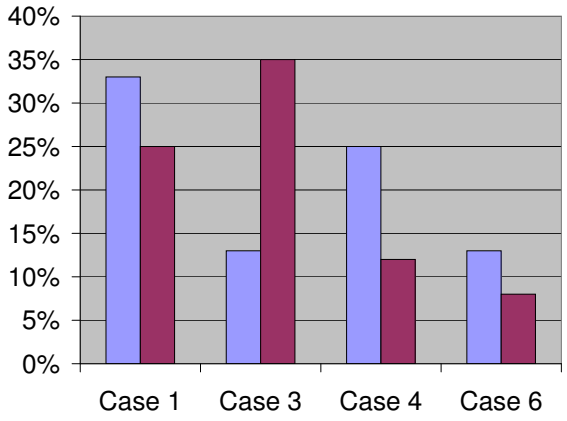

Figure 7. Modification Ratios

In each of the four cases the hot spots indicate an evolution-related issue between pairs of subsystems. The bars in Fig-

ure 7 give the answer to the following question: In how large a percentage of the cases did a modification of a Building Block from a subsystem induce a modification of a Building Block in the other subsystem? Figure 7 makes it more clear that the subsystems from each of the four cases did co-evolve together relatively frequently. This fact may point out that the current implementations of the Building Blocks are not according to the original design decision.

5.4 Evolutionary Cluster vs Single Co-evolution

For each of the top 10 hot spots the underlying evolutionary clusters contained Building Blocks from two different subsystems. During the discussions with the architect it turned out that in every case the Building Blocks were contributing to the same feature.

Relating the co-evolutions crossing subsystem borders may help us to understand evolution-related problems on a more abstract level. This is not possible with the current history-based approaches, mainly because they focus on the individual co-evolutions only. Also, assessing the archive of a software system with evolutionary clusters gives the opportunity to understand the scale of the evolution-related problem better. This is again hard to achieve with the earlier history-based approaches.

6 Discussion and Future Directions

During the development and implementation of our approach we learned about the following issues which need additional research:

- It is important to carefully select the appropriate formulas, see for instance Jaccard similarity or average linkage. Without investigating the effects of the applied formulas, the result of our approach would be meaningless.

- Active interaction with experts is needed in some of the steps of our approach. For instance, when it comes to the validation of the potential issues or when the input for recovering change sets has to be cleaned.

- A deeper knowledge about the development process is needed to customize our approach. When change sets are identified, for example, we had the assumption that every modification related to the same reason is checked-in by the same developer. This might not be true for every development process. Also, the extent to which some change set approximation introduces noise depends on the applied development process.

- Massaging the real world data to fit the applied methods is important. For example, Building Blocks had to be considered in our case study instead of files due to scalability issues. Now that potential issues are identified at the level of Building Blocks, we have to dig deeper to analyze dependencies at the file level.

- Although the cohesion values of the interesting clusters are low, compared to each other they are helpful to point out which part of the software archive has a more severe evolution problem. This way, our approach can be used at the subsystem level to indicate which part of the Initial Partition needs to receive further attention. When we know where to focus, our approach can be re-applied at a more detailed level to observe which lower level software entities play a role in hindering sound evolution.

- The dendrogram visualization of the non-trivial evolution clusters makes it easy to identify hot spots: they are the bottom leaves of the dendrogram which are easy to find, see for instance Figure 5.

Next to the identification of potential evolution-related issues the dendrogram created in our approach can be used for other types of analysis as well. The visualization of the dendrogram helps to spot parts of the archive which evolved different from the rest (see middle part of Figure 5). As can be seen from Figure 5, most of the clusters are joined at a rather low cohesion level. It let us conclude that most of the modifications at the Building Block level are localized.

Although this article describes the assessment of the Initial Partition, our approach can easily be generalized to help create or refine the Initial Partition itself.

In some cases, it might not be possible to create the Initial Partition, because the archive is poorly structured, and there is a lack of expert knowledge available, for instance. Identifying the cluster hierarchy of co-evolving files, as described in our approach, and cutting the dendrogram at a given cohesion level then results in a partition which might be used as an indication of how the archive should be divided. In other words, our approach cannot only be used for assessment purposes but also to provide a partition to start with.

Furthermore, the identified cluster hierarchy can also be useful when there are files we cannot place into the parts. We may group these files into a *default* part of the Initial Partition. Using the cluster hierarchy from our approach, we can observe: (1) how the default part should be divided (2) how parts of the default part should be joined to other parts.

Figure 8 depicts the described generalizations. $P_i, i \in [1..10]$ and DP are parts of the Initial Partition, where DP denotes to the default part. $C_j, j \in [1..6]$ are clusters of

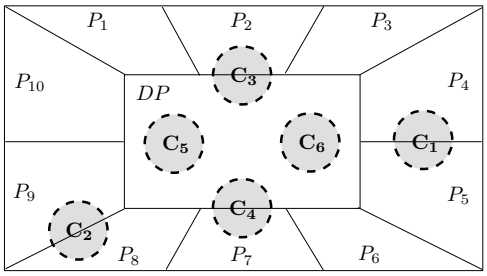

Figure 8. Approach generalization

entities identified by our approach. C_1 and C_2 are used for assessment, as described in this article. C_3 and C_4 can be used to refine the Initial Partition and decide which part of the default part should be assigned to the $P_i, i \in [1..10]$ parts. Further, clusters C_5 and C_6 suggest how to partition DP when no other input is available for the separation.

Although our approach stops at the point of assessing the parts of the Initial Partition, a way to resolve problems is still needed to improve the evolvability of the parts. What types of possibilities are there to do so is also a topic of our future work. We expect that by investigating underlying reasons for the frequent co-evolution of files, for instance call or include relations, we may classify the solution types.

7 Conclusion

Dividing the archive of a legacy system into parts which can evolve as independent as possible is a key evolvability issue. Although this issue was already identified before, most previous work addressed it by looking at static relations (e.g. call relations, include relations) in the source code. However, using static relations can cope with part of the problem only. Software entities can frequently co–evolve even if they are not directly connected by a static relation. To complement previous work, we elaborated how history information, stored in version management systems, can be used to assess a partition of the software archive where the parts are expected to evolve independently in the future.

History information was already used in previous studies, but mainly for identifying patterns in the evolution of software entities and determining modification similarities. We used history information with the intention to create a partition of the software archive such that evolution is as localized as possible.

In this paper, we described our history–driven approach to identify independently evolvable parts of the software archive. Furthermore, we applied our approach in a real industrial environment, considering a software system with multi million lines of code and a long development history. We described both how we implemented our approach and the lessons learned during the application. The application of our approach shows that it can be used effectively to assess the evolving parts of the software system.

References

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing (PRDC '06)*, pages 39–46, Washington, DC, USA, 2006. IEEE Computer Society.

[2] N. Anquetil and T. C. Lethbridge. Recovering software architecture from the names of source files. *Journal of Software Maintenance*, 11(3):201–221, 1999.

[3] J. Estublier, D. Leblang, A. van der Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber. Impact of software engineering research on the practice of software configuration management. *ACM Trans. Softw. Eng. Methodol.*, 14(4):383–430, 2005.

[4] M. Fischer, J. Oberleitner, J. Ratzinger, and H. Gall. Mining evolution data of a product family. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, 2005.

[5] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE '03)*, pages 13–23, Washington, DC, USA, 2003. IEEE Computer Society.

[6] T. Gîrba, S. Ducasse, and M. Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM '04)*, pages 40–49, Washington, DC, USA, 2004. IEEE Computer Society.

[7] T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution*, pages 113–122, 5-6 Sept. 2005.

[8] O. Greevy, S. Ducasse, and T. Girba. Analyzing feature traces to incorporate the semantics of change in software evolution analysis. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05)*, pages 347–356, Washington, DC, USA, 2005. IEEE Computer Society.

[9] M. Jaring, R. Krikhaar, and J. Bosch. Representing variability in a family of MRI scanners. *Software - Practice and Experience*, 34(1):69–100, 2004.

[10] A. Kuhn, S. Ducasse, and T. Gîrba. Enriching reverse engineering with semantic clustering. In *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE '05)*, pages 133–142, Washington, DC, USA, 2005. IEEE Computer Society.

[11] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution - the nineties view. In *Proceedings of the 4th International Symposium on Software Metrics (METRICS '97)*, pages 20–32, Washington, DC, USA, 1997. IEEE Computer Society.

[12] M. Lungu and M. Lanza. Exploring inter-module relationships in evolving software systems. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR '07)*, pages 91–102, Washington, DC, USA, 2007. IEEE Computer Society.

[13] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings of the 6th International Workshop on Program Comprehension (IWPC '98)*, page 45, Washington, DC, USA, 1998. IEEE Computer Society.

[14] O. Maqbool and H. Babri. Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering*, 33(11):759–780, 2007.

[15] A. Mehta and G. T. Heineman. Evolving legacy system features into fine-grained components. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*, pages 417–427, New York, NY, USA, 2002. ACM Press.

[16] R. W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proceedings of the 13th international conference on Software engineering (ICSE '91)*, pages 83–92, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[17] R. C. Tryan. *Cluster Analysis*. Edwards Brothers, Ann Arbor, 1939.

[18] P. van de Laar, P. America, J. Rutgers, S. van Loo, G. Muller, T. Punter, and D. Watts. The Darwin project: Evolvability of software intensive systems. In *Workshop on Evolvability at International Conference on Software Maintenance*, pages 48–53, 2007.

[19] F. J. van der Linden and J. K. Müller. Creating architectures with building blocks. *IEEE Software*, 12(6):51–60, 1995.

[20] L. Voinea, A. Telea, and J. J. van Wijk. CVSscan: visualization of code evolution. In *Proceedings of the 2005 ACM symposium on Software visualization (SoftVis '05)*, pages 47–56, New York, NY, USA, 2005. ACM.

[21] A. Wierda, E. Dortmans, and L. L. Somers. Using version information in architectural clustering - a case study. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR '06)*, pages 214–228, Washington, DC, USA, 2006. IEEE Computer Society.

[22] T. A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, page 33, Washington, DC, USA, 1997. IEEE Computer Society.

[23] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proceedings of the 1st International Workshop on Mining Software Repositories (MSR '04)*, Edinburgh, UK, 2004.

[24] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Trans. Software Eng*, 31(6):429–445, 2005.