Characterizing Evolutionary Clusters

Adam Vanya and Steven Klusener
Computer Science Department
VU University Amsterdam
Amsterdam, The Netherlands
{*vanya,steven*}*@cs.vu.nl*

Nico van Rooijen
CTO MR Systems
Philips Healthcare
Best, The Netherlands
nico.van.rooijen@philips.com

Hans van Vliet
Computer Science Department
VU University Amsterdam
Amsterdam, The Netherlands
hans@cs.vu.nl

***Abstract*—Software architects regularly have to identify weaknesses in the structure of software systems. Groups of software entities which frequently changed together in the past are one way to help find such structural weaknesses. However, there may be many such groups. Not all of them point to structural weaknesses and even fewer indicate severe issues. In this paper we discuss how a multi-dimensional characterization of evolutionary clusters can help identify severe structural weaknesses. In addressing this question we describe (1) properties used for characterizing evolutionary clusters, (2) scenarios characterizing severe structural issues, and (3) the mapping of such scenarios to queries on a set of evolutionary clusters, resulting in a subset denoting severe structural issues according to that scenario. We apply the proposed characterization to the case of a large embedded software system having a development history of more than a decade.**

I. Introduction

In [1], several researchers give their opinion about the future of mining software archives. One of them, Michael Godfrey, is of the opinion that "the future of mining software repositories (MSR) lies in tying software development to the kind of sensemaking that managers and software developers perform daily, right now mostly on the basis of a 'gut feeling.'" And Martin Robillard states "Software developers and other stakeholders spend a lot of time searching for information to solve problems ...". This article fits both these characterizations. We deal with an issue that software architects regularly are faced with: identifying weaknesses in the structure of the software system.

As pointed out by Gall et al. in [2], software entities which changed together in the past can help find such structural weaknesses. Supplementing this observation, Antoniol et al. [3] described how to detect *groups* of software entities changing together. For instance, software entities from subsystems that one would like to evolve independently, and are yet often changed together, might identify such a weakness. The number of such groups, however, may be very large. Not all of them need to point to structural weaknesses, and even fewer indicate *severe* issues. It can therefore be difficult and time consuming to find the few groups of software entities which point to severe structural issues.

When we refer in this article to groups of files changing together, we will use the phrase *evolutionary cluster* to stay consistent with our previous work [4]. The identification of evolutionary clusters is based on the concept of change sets. Change sets contain modifications of software entities, like files or modules, which are related to the same motivation. Such a motivation can be the modification of a feature or the resolution of a problem report. As change sets are not always captured, they often have to be approximated from the available historical information, like check-ins in version management systems. Previous work [3], [5]–[9] describe these approximation techniques. The approximated change sets are then used to derive the evolutionary clusters. In [4] we used a hierarchical clustering algorithm to identify evolutionary clusters. Other known algorithms are using dynamic time warping [3], concept analysis [10] and a visual approach [9] to identify evolutionary clusters.

A common property of the algorithms above is that they all tend to identify a large number of evolutionary clusters. Unless the relevant attributes of these evolutionary clusters are captured and presented, it is difficult to select the evolutionary clusters which indicate high severity structural issues. As for the identification of structural weaknesses, current approaches [9], [11], [12] suggest to look at the clusters which (1) contain software entities from different subsystems and where (2) the entities where modified many times in the recent past. These two criteria seem to be somewhat simplistic. First of all, those two criteria need not fully capture an architect's notion of severity. Second, previous work does not take into account the fact that different architects may have different concerns and therefore consider different weaknesses to be severe.

Furthermore, once we identified evolutionary clusters, it is expedient to carefully categorize those clusters, and to retain this categorization. The knowledge of evolutionary cluster characterization can then be reused when a new state of the software system is to be assessed.

In this paper we present how a carefully prepared characterization of evolutionary clusters can help identify severe structural weaknesses. To address this question we elaborate on (1) which properties of evolutionary clusters could be used for an initial characterization, (2) what architects consider to be a severe structural issue and how this knowledge

can be accessed, (3) how descriptions of severe structural weaknesses can be translated to queries on the evolutionary clusters characterized.

The remainder of this paper is organized as follows. Section II describes the study environment from which we take our examples and experience. Section III elaborates on why it is relevant to characterize evolutionary clusters. Section IV describes the properties used for characterization. Section V briefly describes the concept of evolution anti-scenarios, which capture the knowledge of software architects about what has to be considered as a severe structural issue. Section VI goes through the characterization properties and justifies why they should be used in the characterization. Section VII provides two examples of how the characterization can be used in a real-life situation. Section VIII presents related work. Section IX concludes this paper.

II. Study Environment

To support and validate our results we identified and characterized evolutionary clusters of a large and complex embedded software system. The system studied contains more than 34 000 files comprising eight million lines of code. Hundreds of developers are engaged in the maintenance and development of the system from three development sites, located in different continents. The complexity of the software system has increased over time and handling this complexity has become a challenge.

Programming languages used to implement the system include mainly C#, C++ and C. To identify evolutionary clusters we used historical information on file modifications, like check-in meta-data, from the last nine years of development. We extracted the historical information from the ClearCase version management system.

When evolutionary clusters were identified we had to choose the level of abstraction for software entities. We decided to observe the co-evolutions of *building blocks* because architects were primarily interested in investigating structural weaknesses at that abstraction level. Building blocks are the directories representing the next level of abstraction above individual files in the file hierarchy; see also [13].

In the software system studied architects were interested to know structural weaknesses related to the following types of system decomposition:

- subsystem decomposition
- development group decomposition
- release group decomposition
- deployment group decomposition
- development site decomposition

The last four decompositions listed above are different abstractions above the subsystem level. Development groups are only allowed to modify the subsystems they are responsible for. Release groups contain collections of subsystems which should be released independently. Deployment groups comprise collections of subsystems which should be deployed to different pieces of hardware. Finally, subsystems form groups based on the development site they are developed at.

The approach we have used to identify evolutionary clusters is based on a hierarchical clustering algorithm and is described in [4]. This specific algorithm though has no influence on the message and results of this paper.

During our research we frequently interacted with software architects and software engineers. We especially interacted with a lead architect with whom we had formal meetings nearly every week. That lead architect was actively involved in nearly every step of our research, including: the identification and justification of evolutionary cluster properties (Sections IV, VI), the extraction of evolution anti-scenarios (Section V), translation of those scenarios to queries and the deeper analysis / validation of the evolutionary clusters selected (Section VII).

III. Need For Characterization

Software architects have many tasks to perform [14]–[16]. These tasks include, amongst others, the communication with stakeholders, the translation of requirements to design decisions, and the documentation and assessment of the software architectures developed. Making sure that the subsystems identified can evolve as independent as possible to enable evolvability, is one of the requirements architects often need to consider. Architects typically are pressed for time and have limited time available to resolve structural weaknesses. During this limited time, architects seek to address the most severe structural weaknesses.

To determine how severe a structural weakness is, architects have to weigh the cost of leaving the structure untouched against the cost of resolving the structural weakness. A number of drivers influence these costs; the various dimensions of the characterization we discuss in the remainder of this article are important cost drivers considered by the architects. In case of the observed software system, architects collect high severity structural weaknesses into a list. This list is used to initiate refactoring/restructuring activities. For practical reasons, the list only contains 20 elements. This does not mean architects do not encounter more weaknesses; they just put a threshold on the length, so that only a few weaknesses make it to the list.

Although in practice it is only feasible to resolve a few structural weaknesses, approaches described in the literature typically result in hundreds or even thousands of evolutionary clusters [3] crossing the borders of subsystems. This is not a surprise when we keep in mind that a large software system contains tens of thousands of files. A need therefore arises to filter out the evolutionary clusters which indicate low severity structural weaknesses.

In a large software company it is customary that many people are concerned with the structure of the software system developed. Different architects and developers have different responsibilities and therefore they each define the severity of structural weaknesses in a different way. Architects need to have a global view of the software system and they are usually interested in structural weaknesses from all over the system. Developers on the other hand often need to be more focused and they consider structural issues to be severe if they are related to the subsystems they are working on. Architects themselves are also different. One architect may be interested in structural issues involving different development groups. Another architect may look for frequent co-evolutions between subsystems that are supposed to be released independently. So the severity of structural weaknesses depends on the viewpoint taken.

In order to express which evolutionary clusters are the most important/severe ones for a specific architect or developer, we first need to characterize each of the evolutionary clusters. Previous work [3] implicitly uses a characterization based on the number of times files changed together and the subsystem location of those files. This type of characterization seems to be too simplistic to properly express the level of severity. We know from our practical experience (see also Section VII-C) that such a simple characterization will point to structural weaknesses which are, most of the time, not acknowledged by the architect to be of high severity. Unless we properly characterize the evolutionary clusters we run the risk that the added value of the evolutionary cluster identification is deemed limited by industry.

IV. An Evolutionary Cluster Characterized

Characterization of evolutionary clusters involves identifying the properties of those clusters and measuring the actual values for those properties. Different evolutionary clusters can have different values for those properties. In this section we elaborate on the characterization of one real-life evolutionary cluster. Why exactly the properties described below are the ones which we propose is described in Section VI. The evolutionary cluster we characterize here was identified using the dendrogram approach [4] with a 200s sliding window parameter [17].

A. Property 1: Cluster Size

The size of an evolutionary cluster is the number of software entities involved. The size therefore may be an indication of its complexity and changeability. The bigger the size, the more entities need to be changed together and the more complex the whole operation might be. In our case the software entities were building blocks, which are the next abstraction level above files. Each building block contains 50 files on average. In our example the evolutionary cluster contained seven building blocks.

B. Property 2: Borders Crossed

Every software entity in the evolutionary cluster can be identified using different decompositions. In our example, every building block has its containing subsystem, development group, deployment group, release group and development site. A development group owns one or more subsystems and is responsible for the modifications introduced to those subsystems. A deployment group contains subsystems which need to be deployed independently. The independent release group consists of subsystems to be released independent of the rest of the system. As the observed industrial environment is multi-site, the building blocks are developed in different parts of the world.

The observed evolutionary cluster contains building blocks from two different subsystems and therefore the cluster crosses the borders of subsystems. These subsystems were part of different release groups, so the evolutionary cluster crosses the borders of independent release groups. As for the other decompositions, the observed cluster is located in a single part.

C. Property 3: Support Distribution

The support between two entities, in our case building blocks, shows how many times those entities changed together, see [18]. This property indicates, for a given system decomposition, the distribution of the support values for all the software entity pairs in the cluster that come from different decomposition parts. If we are interested in subsystem decomposition, then a decomposition part is a subsystem; if we are interested in the development group decomposition, then a decomposition part is the set of subsystems owned by a development group.

In our example, we are interested in the subsystem decomposition. The evolutionary cluster observed contains subsystem crossing relationships with a relatively high support, see Table I.

Table I
Support Distribution

Support			
MAX	MIN	AVG	STD
107	32	66	21

D. Property 4: Confidence Distribution

If two software entities E_1 and E_2 changed 7 times together and 3 times separately, then the confidence of their relationship is 0.7, or 70%. The confidence expresses the probability of two software entities changing together given that one of them gets changed, see also [18]. This property measures, for a given system decomposition, the distribution of the confidence of software entity pairs which come from different decomposition parts.

To analyze the distribution, we measure the maximum, minimum, average and the standard deviation of the confidences. For the subsystem decomposition the results are depicted in Table II.

Table II
CONFIDENCE DISTRIBUTION

Confidence			
MAX	MIN	AVG	STD
24.71%	8.29%	15.42%	4.52%

E. Property 5: First Co-evolutions

We cannot only measure how many times software entities changed together, but we can also observe when exactly they changed together for the first time. The date of the first co-evolution helps to understand since when the participating software entities are related. This property captures, for a given system decomposition, the distribution of the first co-evolutions between software entities from different decomposition parts. The first co-evolutions are indicated with the letter **F** in Figure 1.

The actual data for the subsystem decomposition is included in Table III, showing the maximum, minimum, average and the standard deviation(given in days) of the first co-evolutions. As we can see, all building blocks that participate in the analyzed evolutionary cluster were first changed together on the same day. This statement happens to be true for our example; it is not always the case.

Table III
FIRST CO-EVOLUTION DISTRIBUTION

First Co-evolution			
MAX	MIN	AVG	STD
15 July 2004	15 July 2004	15 July 2004	0 days

F. Property 6: Last Co-evolutions

Similar to Property 5, this property expresses, for a given system decomposition, the distribution of the *last* co-evolutions between entities from different decomposition parts. The last co-evolutions are indicated with the letter **L** in Figure 1. Table IV shows the actual distribution values for the subsystem crossing relationships in case of the evolutionary cluster studied.

Table IV
LAST CO-EVOLUTION DISTRIBUTION

Last Co-evolution			
MAX	MIN	AVG	STD
30 Apr 2009	19 Aug 2008	28 Jan 2009	104 days

G. Property 7: Co-evolution Tendencies

Co-evolutions between two software entities have a certain distribution over time. This distribution may show different tendencies: (1) co-evolutions getting more frequent, (2) co-evolutions getting less frequent, or (3) co-evolutions having a more or less stable frequency of occurrence over time.

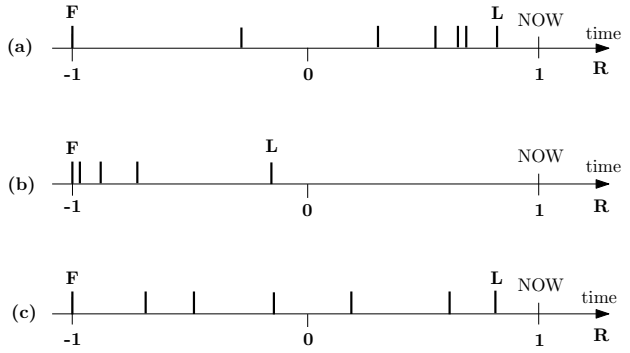

Figure 1. Co-evolution Tendency

Figure 1 indicates these three main types of tendency. We determine the tendency type by mapping the period between the first co-evolution and NOW on a [-1,1] interval, see Figure 1, and we determine the position of every co-evolution in this interval. The average of all the resulting numbers is referred to as the *tendency number*. A tendency number near to -1 tells us that the co-evolution frequency decreased, a tendency number of near 1 shows that co-evolutions getting more frequent and a 0 value indicates that co-evolutions are evenly distributed. This property describes, for a given system decomposition, the distribution of the tendency numbers between software entities from different decomposition parts. Table V shows the maximum, minimum, average and the standard deviation for the tendency values for the subsystem crossing relationships.

Table V
CO-EVOLUTION TENDENCY DISTRIBUTION

Co-evolution Tendency			
MAX	MIN	AVG	STD
-0.16	-0.30	-0.23	0.04

H. Property 8: Static Relationships

With this property we count, for a given system decomposition, how many entity pairs from different parts are also coupled in terms of static relationships. Static relationships can be include relations or call relations, for instance. These relationships can help to identify couplings between entities by just having a look at the content of those entities. In

case of our example we found seven static relations crossing subsystem borders.

I. Discussion

The assessment of the severity of the evolutionary cluster studied in this section depends on one's interests:

- The building blocks in the evolutionary cluster were changed relatively recently. This might suggest they will again change in the near future.
- The building blocks all come from the same development group, so from that point of view the cluster is not severe.
- The size of the cluster (7) is fairly large, compared to other clusters. From the size perspective, this cluster definitely indicates a severe issue.

V. Evolution Anti-Scenarios

The characterization described in Section IV can be applied to all the evolutionary clusters. As a result, we know for every evolutionary cluster exactly what properties they have. For instance, we know which clusters are crossing the borders of subsystems and which are the clusters where the subsystem crossing relationships have a high average support.

When evolutionary clusters indicating *severe* structural weaknesses are to be selected, we have to define a query on the properties identified. We can specify, for example, that we are interested in evolutionary clusters (1) crossing the borders of development groups, (2) having a high support distribution for the relationship crossing the borders of development groups, and (3) where the building blocks from different development groups have co-evolved much more frequently in the recent past than earlier in time.

To define our query to select evolutionary clusters, it has to be clear what types of structural weaknesses are severe (i.e. costly). What is considered severe depends on the architects or developers and their responsibilities. Once we know which structural weaknesses experts are looking for, we should be able to query evolutionary clusters that point to those weaknesses. Architects however are not thinking explicitly in terms of evolutionary cluster properties when severe weaknesses are described. According to our experience, architects tend to express what they consider severe in terms of concrete scenarios. These scenarios are often coming from their past experience and are described in a domain specific way. Using scenarios is a known way to assess properties of software architectures [15]. The next paragraph describes one such scenario.

Developer D_A is a member of the development group G_A. G_A owns subsystem *A* and is responsible for introducing modifications to that subsystem. In a similar way, we have developer D_B from development group G_B owning subsystem *B*. One day, D_A changes a set of files and realizes that, as a consequence, other files from subsystem *B* also need to be changed. So he calls D_B at the other side of the world and asks him to introduce some modifications. As G_B is busy implementing other functionalities, D_B puts the request onto a priority list. After 10 telephone calls from D_A and after one month of time has elapsed, the required files in subsystem *B* are modified. Altogether it took 40 man-hours to introduce the change initiated by D_A. Changes which are similar in complexity but involve only G_A cost usually 4 man-hours.

The above scenario is an example of what we call an *evolution anti-scenario*, analogous to the notion of design anti-patterns [19]. Such a scenario describes what architects do *not* want to happen during the evolution of the software system. Evolution anti-scenarios are usually kept implicit in the heads of the architects. Interviews can be used to elicit the implicit knowledge of anti-scenarios. During these interviews, documents describing which structural weaknesses an architect resolved or plans to resolve may help to get (or derive) the explicitly described anti-scenarios. The extracted anti-scenarios can then be used to understand which type of structural weaknesses the architects and developers consider to be severe. Understanding which structural weaknesses are severe is the result of an iterative approach rather than the result of a one shot activity.

VI. Property Justification

Having the evolution anti-scenarios collected we need to translate them to one or more queries on the collection of evolutionary clusters. This puts an additional requirement on the characterization. The properties (size, borders crossed, etc.) we use to characterize clusters should allow us to query evolutionary clusters and identify the types of structural weaknesses described by the scenarios. In this section we will argue, for every property described in Section IV, that they should be included in the characterization. These properties need not form a complete set. In this section, we use real-life evolution anti-scenarios from the study environment to support our arguments.

A. Property 1: Cluster Size

Let us consider the following scenario. In a large software development company it is quite common that experienced developers are leaving, while others with little or no domain knowledge are entering the organization. Developer *D* is such a newly employed developer. *D* gets involved with his new project and his project leader assigns *D* the task to modify the database schema. *D* modifies the database schema and some other software entities which he thinks are affected by the change. Being new and having a lack of domain knowledge, however, *D* forgets to modify another ten software entities from different parts of the software structure. As a consequence, the test of the software system fails. Such a scenario costs the company a lot of effort (in time and money).

The lesson we can learn from the above evolution anti-scenario is that the more software entities are involved, the more difficult it is to maintain consistent changes. If the project leader knows the size of evolutionary clusters, he may assign a task to *D* which involves smaller evolutionary clusters and therefore less complexity.

Huge evolutionary clusters, however, are problematic even for experienced developers. Experts could therefore select huge evolutionary clusters and try to reduce their size by making the involved software entities less dependent.

On the other hand, in some cases small evolutionary clusters are the ones which are important. For instance, if the goal is to resolve as many structural issues as possible from a fixed budget, then evolutionary clusters which indicate cheap-to-resolve structural weaknesses are probably the target for the resolution activities. The effort to resolve a structural issue is likely to be influenced by the size of the evolutionary cluster. The bigger the evolutionary cluster is, the more entities are related and the more effort it may take to resolve the structural weakness related.

B. Property 2: Borders Crossed

Let us reconsider the evolution anti-scenario described in Section V. In that scenario, software entities from different development groups and development sites had to be changed together. Compared to the changes which are local to development groups, these types of changes tend to be much more costly. The increase in costs is caused by the additional communication costs and the costs caused by the fact that one development group has to wait for another one.

Certain architects may be responsible for reducing the above costs by carefully assigning software entities to development groups. In order to assess and to improve the current structure, these architects have to know which entities from different development groups changed together. In this case, the architects in charge are less interested in evolutionary clusters not crossing the borders of these groups. If an evolutionary cluster is not crossing a border of development groups it may still cross the borders of independent release groups, for instance. Those evolutionary clusters in turn may be relevant for architects responsible for the release group structure.

C. Property 3: Support Distribution

A description of an evolution anti-scenario might be as follows. In a software company new projects were started last year, to develop new releases of their software product with improved and new features. Of course, during this period certain files were modified much more frequently than others. We observed that one set of files FS_1 from subsystem S_{FS_1} changed together with a set of files FS_2 from subsystem S_{FS_2} 20 times during the last year. With this number of co-evolutions, FS_1 and FS_2 are the file sets which changed together most often in the last year.

In the above scenario the co-evolution of FS_1 and FS_2 may indicate a severe structural weakness. Even if changing those sets of files together is not the most costly operation, changing them together 20 times makes the underlying structural weakness the most costly one during the last year of development. In case development activities will touch upon the same part of the software system next year, architects may want to resolve structural weaknesses similar to the one indicated by the co-evolution of FS_1 and FS_2. Usually, architects look for the outliers when considering the support distribution by adapting the organization.

D. Property 4: Confidence Distribution

It is not uncommon that after a while some component of a long-living software system needs to be replaced by another. Usually the new component provides at least the functionality of the replaced component, while also bringing additional benefits. For instance, the development of a component can be outsourced to fulfill extreme performance requirements and/or to decrease development costs.

Let us consider the following, related, scenario. A decision has been taken by the architect that subsystem S needs to be outsourced and his decision is approved by management. The decision was taken in order to reduce development costs. Company C_O takes over the development of S. Although S is outsourced now, it is still actively connected to the rest of the system: very often, a change in S requires a change in the rest of the system and vice versa. After a year of co-operation it turns out that the communication and the increased management costs due to the poor isolation of S exceeded the original development costs. Consequently, a decision is taken to in-source S again.

When looking for severe structural weaknesses between the outsourced subsystem and the rest, the relative distribution of single-site changes as opposed to site-crossing changes, i.e. the confidence distribution, may play a role. If, relatively speaking, the number of local changes is large (i.e. confidence is low), this may warrant keeping the subsystem outsourced, for financial reasons, or because of local expertise. If, on the other hand, the confidence is high, communication and collaboration cost may become a bottleneck, and one may decide to resolve this weakness.

E. Property 5: First Co-evolutions

The first co-evolution between two software entities determines since when they were modified together. Files evolving together since last month may indicate a less severe structural weakness than files which evolved together during the last four years. Four years of continuous co-evolution tells us that the reason why they are changing together is still not resolved. On the other hand, files which only changed together recently may have done so, for instance, because of the introduction of a new feature. In the software system we observed it happens frequently that after a new feature

introduction, it takes a while before the concerns are well separated. Co-evolution during that initial period need not indicate a severe issue.

F. Property 6: Last Co-evolutions

During the development and maintenance of a software system structural weaknesses are continuously resolved. As a result we often see from the development history that files changed together a long time ago but after a while stop doing so. Therefore, evolutionary clusters containing files where the last co-evolutions happened a long time ago probably indicate (1) an already resolved structural issue or (2) stable couplings between files with respect to evolution. Therefore, those type of evolutionary clusters are not interesting to the architects and need to be filtered out.

G. Property 7: Co-evolution Tendencies

Files which co-evolved a lot but the frequency of co-evolution has become very low may not be interesting to the architects because at present those files are less likely to change together. On the other hand, files which have been modified together periodically, let's say every month, will most probably change together in the next month if we do not do anything against it. Therefore it is important to measure the tendencies of co-evolution frequencies between software entities.

H. Property 8: Static Relationships

One evolution anti-scenario runs as follows. It becomes important to separate the evolution of subsystem S_1 from the evolution of subsystem S_2. To reach that goal, evolution-type dependencies between S_1 and S_2 are identified using static relationships only. Static relationships are easy to determine from the source code. According to our experience, static relationships are also the kind of relationships which software developers tend to recall, and resolve, most easily. But resolving static relationships is not enough. The anti-scenario we are considering here concerns a situation where we are stuck with an evolutionary cluster without any underlying static relationships, since that is more likely to point to an issue yet unknown.

VII. Querying Evolutionary Clusters

In order to find evolutionary clusters which point to severe structural weaknesses we need to create and execute a query on the set of evolutionary clusters characterized. Such a query has to result in evolutionary clusters where the corresponding structural weaknesses are considered severe by a specific architect or developer. To create the query, we use as input the evolution anti-scenarios, see Section V, elicited from the architect or developer we want to support. We now describe two real-life cases when the elicited evolutionary scenario(s) were translated to queries.

A. Case 1

In the first case, the architect had the task to make sure that the development groups, owning specific parts of the software system, depend as little as possible on each other. Evolution anti-scenarios related to this case describe (1) a large amount of communication between development groups and (2) delay of work of some development groups by that of others. Based on these scenarios we created the query on the evolutionary clusters with the criteria described in Table VI.

Table VI
Selection Criteria

PROPERTY	CRITERION
Cluster Size	≥ 3
Borders Crossed	= "Development Groups"
Support AVG	> 50
Confidence AVG	$< 20\%$
First co-evolution MIN	more than 2 month ago
Last co-evolution MAX	less than 2 years ago
Co-evolution tendency AVG	$>$ -0.15
Static relations	–

When a change involves more than one development group, the number software entities involved, and therefore the complexity of the change, does have an influence on how much communication effort is spent. Furthermore, more files typically take more time to modify and in such cases development groups may delay others for a longer time. Therefore we selected evolutionary clusters where there is more than just two entities.

It was straightforward to select the evolutionary clusters crossing the borders of development groups since the architect we served was interested in the weaknesses of the development group structure and not in the weaknesses of other structures. A support threshold was included in the query because we wanted to identify structural weaknesses indicating that different development groups need to communicate frequently. The more frequently development groups need to communicate, the more frequently the effort on communication needs to be spent. The date thresholds for the first and last co-evolutions were set such that resulting evolutionary clusters indicate weaknesses with the development group structure which are not yet resolved. To give priority to structural weaknesses which make the developers from different groups communicate more and more often we set a threshold for the tendency number.

As we can see, there was no filtering criterion set on the static relations. The reason for that is that the architect wanted to have an overview of the severe structural weaknesses independent from whether they are easy to identify from the static relationships or not.

B. Case 2

In the second case we consider an architect who has the task to make release groups more independent. The final goal is to release new versions of every release group on their own, without depending on one another. As a first step, static relationships were analyzed to know at which points release groups are coupled. This step is thought to be relatively cheap and effective, but looking only at static relationships does not allow us to identify all the severe structural weaknesses. Therefore the architect is interested in structural weaknesses where the unwanted co-evolutions are not caused by direct static dependency. The evolution anti-scenario in this case describes that components from different release groups cannot be released independent although static dependencies are removed. Based on these scenarios we created the query on the evolutionary clusters with the criteria described in Table VII.

Table VII
SELECTION CRITERIA

PROPERTY	CRITERION
Cluster Size	–
Borders Crossed	= "Release Groups"
Support AVG	> 20
Confidence AVG	$< 20\%$
First co-evolution MIN	more than 2 month ago
Last co-evolution MAX	less than 2 years ago
Co-evolution tendency AVG	> -0.15
Static relations	$= 0$

As compared to the previous case, we had to filter the characterized evolutionary clusters in a different way. First, we considered only the evolutionary clusters where there were no static relationships between the entities included. Second, this time not the development groups but the release groups were of interest. Third, we defined no restriction for the cluster size and we specified a lower threshold for the support than in the previous case. The reason is that co-evolutions without static dependencies are much more costly than the ones caused by static dependencies. Also, there are much less structural weaknesses caused by static dependencies. Consequently for the architect it is important to know of cases even if the entities included evolved less frequently together and even if the cluster contains fewer entities.

C. Discussion

Executions of the queries described in the above two cases result in subsets of the evolutionary clusters identified. Related work considers an evolutionary cluster to indicate a structural issue if it crosses subsystem borders and if the related software entities changed many times together, see [3]. This definition suggests to take only the number of co-evolutions in an evolutionary cluster as an indication for *how* severe the related structural issue is. Using this default prioritization Table VIII indicates in both of the above cases, at which positions the evolutionary clusters selected would end up.

Table VIII
PRIORITY MAPPING

Case 1	Default
1	1
2	2
3	9
4	34
5	36
6	37
≥ 7	>50

Case 2	Default
1	4
2	9
3	19
4	22
5	29
6	37
≥ 7	>50

Table VIII shows that in both cases the top 50 from the default prioritization contained only six evolutionary clusters in which architects were actually interested. So the characterization helped us to select evolutionary clusters which are relevant to the architects supported. In the first case (section 7.1) we went further and analyzed the clusters deeper with a lead architect and a software engineer. Both the architect and the software engineer acknowledged that all the evolutionary clusters selected were not only interesting because of the characterization but also because deeper analysis reveals a severe evolution-type issue.

Again, we want to emphasize that we do not claim to find *the* severe structural issues exclusively. We claim that by using our approach architects can better handle/reduce the huge amount of clusters which previous work generates. Consequently, software architects may need more information (deeper analysis) than just the result of the query to decide upon the severity of an evolutionary cluster.

We expect that in most of the cases one will use the MIN value for the first co-evolution, MAX value for the last o-evolution, and the AVG values for the other properties. However, knowing the distribution of values allows us to detect, and possibly remove, outliers.

It is not guaranteed, however, that the initial set of properties as defined in Section IV is complete enough to express what architects consider to be an evolutionary cluster pointing to a severe structural issue. Therefore, it may be necessary to extend the set of properties used for characterization and/or to have a deeper analysis of the evolutionary clusters selected.

VIII. RELATED WORK

Next to the related work which we have already referred to, there are other related contributions. In this section we give a brief overview of these contributions and we describe how this work is related to the contribution of this paper.

In [20] Treude and Storey describe an interactive tool to visualize how the relevance of concerns in the software system changed over time. Measuring how the relevance of different concerns changed over time is used to identify which concerns did co-occur in a specified time-frame. The tool of Treude and Storey allows its users to define filters on the cluster of concerns identified, somewhat similar to our filtering of evolutionary clusters using the characterization.

Fisher and Gall [12] also identify structural weaknesses. They first filter out the "interesting" file pairs, where interesting means that the files changed frequently together and that those files are from different parts of the system. These interesting file pairs form the input for a visualization to detect anti-patterns, like god-classes. Although the visualization indicates structural anti-patterns, the approach is not addressing explicitly the prioritization of structural issues.

German in [7] characterizes different types of modification requests, what we call change sets. The characterization applied is used to understand the nature of the change sets. Our work takes the characterization further to the level of evolutionary clusters which are identified based on the change sets.

Breu and Zimmermann [21] mine aspects from version histories. The authors define the concepts of simple and complex aspect candidates showing some similarity to what we call change sets and evolutionary clusters respectively. To identify relevant aspects Breu and Zimmermann use size, support and compactness measures as an input for aspect candidate prioritization.

Using visualizations is an alternative way to identify evolution-type structural weaknesses [11], [22]. Typically, one looks for those structural weaknesses by identifying patterns in the visualization. Which pattern instance is selected to be further analyzed depends on ones intuition. Pinzger et al. [23] helps the formulation of such an intuition by visualizing multiple evolution metrics for several releases of the software system.

IX. Conclusion and Future Work

In this paper we have described how to characterize evolutionary clusters and how this characterization may help architects and developers to find severe weaknesses in the structure of a software system. We showed that a well prepared characterization is important to support software architects and developers. We described which properties of evolutionary clusters are to be measured for the characterization. We found that, in practice, evolution anti-scenarios are the main source of information on what is considered to be a severe structural weakness. Furthermore, for every property we have argued that they are important for the characterization. Finally, we argued that the characterization itself may evolve over time and that therefore there is a need to keep it up-to-date.

The properties we identified are very general, and are also found in other systems. Technically speaking, our method is applicable to other systems as well. Whether the properties we distinguish reflect the main cost drivers for the evolution of other systems remains to be investigated.

Characterizing evolutionary clusters and querying them are only the first steps towards the identification of severe structural weaknesses. A further pruning of the set of evolutionary clusters identified may be necessary, since some of them may still point to structural weaknesses which do not turn out to be a severe issue after all. In order to decide whether this is the case we need to analyze the identified clusters at a deeper level of detail. This means that there is a need to explore the clusters to see (1) which are the actual files or methods which changed together, (2) what is the distribution of the properties measured between those software entities (for instance how many times file pairs change together), (3) what is the reason that those software entities changed together. A good visualization supporting the exploration of evolutionary clusters is therefore essential not only to further analyze severity but also to investigate what could cause a structural issue. This is a topic of future work.

The lead architect we contacted frequently during the characterization was positive about the usefulness of the approach described in this paper. Thinking about characterization explicitly helps refining the abstract requirement of independent evolution and to express sharper what a severe evolution-type structural issue is.

It is a recurring task of software architects to modify the decomposition of the software system. In those cases structural issues do not show up immediately and are not reported back from the developers. Architects acknowledged that finding severe structural issues faster in the newly defined structure is one of the major values of using characterized evolutionary clusters. For instance, now it is easier for those architects to assess the new development group decomposition.

Acknowledgment

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

References

[1] M. W. Godfrey, A. E. Hassan, J. Herbsleb, G. C. Murphy, M. Robillard, P. Devanbu, A. Mockus, D. E. Perry, and D. Notkin, "Future of mining software archives: A roundtable," *IEEE Softw.*, vol. 26, no. 1, pp. 67–70, 2009.

[2] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proceedings of the International Conference on Software Maintenance (ICSM '98)*. Washington, DC, USA: IEEE Computer Society, 1998, p. 190.

[3] G. Antoniol, V. F. Rollo, and G. Venturi, "Detecting groups of co-changing files in cvs repositories," in *Proceedings of the Eighth International Workshop on Principles of Software Evolution (IWPSE '05)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 23–32.

[4] A. Vanya, L. Hofland, S. Klusener, P. v. d. Laar, and H. v. Vliet, "Assessing software archives with evolutionary clusters," in *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension (ICPC '08)*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 192–201.

[5] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 429–445, 2005.

[6] 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)*. Washington, DC, USA: IEEE Computer Society, 2003, p. 13.

[7] D. M. German, "An empirical study of fine-grained software modifications," *Empirical Softw. Eng.*, vol. 11, no. 3, pp. 369–393, 2006.

[8] A. T. T. Ying, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Trans. Softw. Eng.*, vol. 30, no. 9, pp. 574–586, 2004.

[9] D. Beyer and A. E. Hassan, "Animated visualization of software history using evolution storyboards," in *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 199–210.

[10] T. Gîrba, S. Ducasse, A. Kuhn, R. Marinescu, and R. Daniel, "Using concept analysis to detect co-change patterns," in *Ninth international workshop on Principles of software evolution (IWPSE '07)*. New York, NY, USA: ACM, 2007, pp. 83–89.

[11] M. D'Ambros and M. Lanza, "Reverse engineering with logical coupling," in *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 189–198.

[12] M. Fischer and H. Gall, "Evograph: A lightweight approach to evolutionary and structural analysis of large software systems," in *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 179–188.

[13] F. J. van der Linden and J. K. Müller, "Creating architectures with building blocks," *IEEE Softw.*, vol. 12, no. 6, pp. 51–60, 1995.

[14] C. Hofmeister, R. Nord, and D. Soni, *Applied software architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[15] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, April 2003.

[16] P. Clements, R. Kazman, M. Klein, D. Devesh, S. Reddy, and P. Verma, "The duties, skills, and knowledge of software architects," in *Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA '07)*. Washington, DC, USA: IEEE Computer Society, 2007, p. 20.

[17] T. Zimmermann and P. Weißgerber, "Preprocessing CVS data for fine-grained analysis," in *Proceedings of the First International Workshop on Mining Software Repositories*, May 2004, pp. 2–6.

[18] T. Zimmermann, S. Diehl, and A. Zeller, "How history justifies system architecture (or not)," in *Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE '03)*. Washington, DC, USA: IEEE Computer Society, 2003, p. 73.

[19] T. Feng, J. Zhang, H. Wang, and X. Wang, "Software design improvement through anti-patterns identification," in *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM '04)*. Washington, DC, USA: IEEE Computer Society, 2004, p. 524.

[20] C. Treude and M.-A. Storey, "Concernlines: A timeline view of co-occurring concerns," in *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering (ICSE '09)*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 575–578.

[21] S. Breu and T. Zimmermann, "Mining aspects from version history," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE '06)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 221–230.

[22] J. Ratzinger, M. Fischer, and H. Gall, "Improving evolvability through refactoring," in *Proceedings of the 2005 international workshop on Mining software repositories (MSR '05)*. New York, NY, USA: ACM, 2005, pp. 1–5.

[23] M. Pinzger, H. Gall, M. Fischer, and M. Lanza, "Visualizing multiple evolution metrics," in *Proceedings of the 2005 ACM symposium on Software visualization (SoftVis '05)*. New York, NY, USA: ACM, 2005, pp. 67–75.