Multidimensional characterization of evolutionary clusters: An experience report

Adam Vanya a,*, Steven Klusener a, Nico van Rooijen b, Hans van Vliet a

a VU University Amsterdam, De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands
b Philips Healthcare, Veenpluis 4–6, 5684 PC Best, The Netherlands

1. 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 in our study environment had to face: identifying unwanted couplings between parts of a software system.

Imagine an embedded system with two hardware components A and B, whose software is decomposed into three major subsystems X, Y and Z. Suppose X and Y are developed at one site, while Z is developed at another site. Furthermore, suppose subsystems X and Y often change together during the evolution of the system, but subsystem Z changes independently from X and Y. This creates a dependency between X and Y. If X is associated with hardware component A, and Y and Z are associated with hardware component B, this situation may be deemed undesirable. The reason for this is that if we want to, say, change to another manufacturer for hardware component A, such may not be easy to accomplish because of the dependency between A’s software (X) and the rest (Y and Z). We call this dependency an unwanted coupling.

This notion of an unwanted coupling is relative: in the above example, it is only an unwanted coupling from the point of view of wanting to be able to change hardware components easily. If we are not concerned about changing hardware components, but only about the amount of communication between development sites, then the above dependency probably would not be considered an unwanted coupling. A dependency between subsystems only is an unwanted coupling in the eye of the beholder.

In the above example, we talked about subsystems which change together. What happens of course is that certain parts of these subsystems (such as components, classes) change together.
We can retrieve that kind of information from the version management system, as illustrated by the seminal work of Ball et al. on change couplings [2].

Software entities (such as files or components) which changed together in the past can help find unwanted couplings [3]. Antoniol et al. [4] describe how to detect groups of frequently co-changing software entities. We refer to those groups of frequently co-changing entities as evolutionary clusters, see also our previous work [5].

The identification of evolutionary clusters is based on the concept of change sets. Change sets contain modifications of software entities, such as files, which are changed because of the same logical reason. Such a reason can be the modification of a feature or the resolution of a problem report. As change sets are not always captured explicitly, they often have to be approximated from the available historical information, such as check-ins in version management systems. Previous work [6,7,4,8–10] describe these approximation techniques. The approximated change sets are then used to derive the evolutionary clusters. In [5] we used a hierarchical clustering algorithm to identify evolutionary clusters. This process is sketched in Section 2. Other known algorithms are using dynamic time warping [4], concept analysis [11] or a visual approach [10] to identify evolutionary clusters.

A common property of most of these algorithms is that they tend to identify a large number of evolutionary clusters. As for the selection of unwanted couplings from the set of evolutionary clusters, current approaches [10,12,13] suggest to look at the clusters which (1) contain software entities from different subsystems and where (2) the entities were modified many times in the recent past. These two criteria seem to be somewhat simplistic. They need not fully capture an architect’s notion of an unwanted coupling, as evidenced by the embedded system example discussed above. We use a larger set of criteria (see Section 5) to characterize evolutionary clusters. In a next step, the architect uses these criteria to express which evolutionary clusters denote unwanted couplings from his point of view. Different architects may have different concerns and therefore consider different couplings to be unwanted. In our approach such is achieved by allowing different architects to choose different values for the various criteria, thereby selecting a different subset of the evolutionary clusters. The set of values for the various criteria is derived from an evolution anti-scenario, a short description of a situation architects do not want to observe during the evolution of the system (see Section 4).

Furthermore, once we characterized which evolutionary clusters denote unwanted couplings, it is expedient to carefully retain this characterization. This knowledge can be reused when a new state of the software system is to be assessed. An example hereof is discussed in Section 7, where we investigate whether moving certain files from one subsystem to another would solve an unwanted coupling, or lead to different unwanted couplings elsewhere. To that end, we redo the characterization on a different subsystem decomposition and compare the outcome with that of the original decomposition. This ‘what-if’ type of investigation allows for a quick assessment of the effects of structural changes to the software archive.

Let us summarize the process we follow. We first approximate change sets – sets of software entities that are changed together for the same reason – from the version management system. Next, we use these change sets to determine evolutionary clusters, of which there can be a lot. These evolutionary clusters are characterized along different dimensions, such as whether they involve more than one site, how many entities are involved, and so on. The particular interest of the architect, i.e. the type of evolutionary cluster he considers to denote an unwanted coupling, is next derived from an evolution anti-scenario which is used to query and filter the set of evolutionary clusters. In a final step, the resulting unwanted clusters are analyzed one by one to understand the underlying reason for the co-change behavior, and determine how to resolve the issue. This last step is discussed in [14].

Architects are under time pressure, so they can only investigate a small subset of the evolutionary clusters. Architects do not want to spend time investigating evolutionary clusters that do not match their current interest. In this paper we present how a carefully prepared characterization of evolutionary clusters can help filter out the unwanted couplings an architect is interested in. To address this question we make the following contributions:

- Elaborate on which properties of evolutionary clusters could be used to characterize the clusters.
- Describe what architects consider to be an unwanted coupling and how this knowledge can be captured in so-called evolution anti-scenarios.
- Discuss how these anti-scenarios can be translated to queries on the set of evolutionary clusters. Executing this query gives the subset of evolutionary clusters denoting unwanted clusters.

The remainder of this paper is organized as follows. Section 2 sketches how change sets are used to derive evolutionary clusters. Section 3 describes the study environment from which we take our examples and experience. Section 4 briefly describes the concept of evolution anti-scenarios, which captures the knowledge of software architects about what is considered an unwanted coupling. Section 5 describes the characterization properties and justifies why they should be used in the characterization. Section 6 provides two examples of how the characterization can be used in a real-life situation. Section 7 discusses a case, where the characterization helped to execute a what-if type of scenario. Section 9 describes the threats to validity. Section 8 summarizes our lessons learned. Section 10 presents related work. Section 11 concludes this paper.

This paper is based on our previous work [15]. In this paper, we have added the ‘what-if’ type of investigation mentioned above, together with an example thereof in Section 7. We have also extended our previous paper with a study comparing the outcome of the different approaches to query evolutionary clusters, see Section 6.3. The outcome of the study further clarifies in which ways it is beneficial to apply the multidimensional characterization of evolutionary clusters proposed.

2. From change sets to evolutionary clusters

To illustrate the clustering algorithm, let us consider an example, involving four entities and six change sets. Table 1 indicates which entities are contained in each change set. The $\rightarrow$ symbol is used to express that a change set contains an entity.

As a first step, we measure the Jaccard similarity between each pair of entities. The Jaccard similarity expresses the probability of two entities changing together given one of them gets changed; see also [16,17]. For the above example, entities $C_1$ and $C_4$ changed twice together (in the first and third change set), while $C_4$ changed once without $C_4$ (in the second change set), and $C_4$ changed three times without $C_1$ (in the fourth, fifth and sixth change set). So
the Jaccard similarity is $2/(2 + 1 + 3) = 0.33$. The complete set of Jaccard similarities is given in Table 2.

The next step is to iteratively cluster entities. We use the Agglomerative Hierarchical Clustering Analysis [18] to do so. We start with sets containing one entity only and we join sets until only one set remains. In each step, we join the two sets with the highest average linkage value. The average linkage value of two sets $A$ and $B$ is the average of the Jaccard similarities $J_{XY}$, where $X$ is an entity from $A$ and $Y$ is an entity from $B$. There are many ways to aggregate entity-level similarities to the level of sets of entities. We chose the average linkage since other linkage methods result in a hierarchy that is very sensitive to slight changes in the strength of entity similarities. A relative stability of results is important if we want to base longer term architectural decisions on the identified clusters.

For our example, we will cluster $C_2$ and $C_3$, since their Jaccard similarity is 0.66. In the next step, we have to consider the three average linkage values given in Table 3. So we next cluster ($C_2$, $C_3$) and $C_4$, with average linkage value 0.5. And in the final step we cluster all four entities, with a resulting average linkage value of 0.251.

We call the identified groups of entities evolutionary clusters. Since we are interested in analyzing frequent changes involving different subsystems, we prune the set of evolutionary clusters by removing all that involve a single subsystem only. The result is a set of evolutionary clusters involving more than one subsystem. Note that we started this step with a collection of change sets, and end with evolutionary clusters which are sets of entities. Further details of this clustering are given in [5].

### Table 2

<table>
<thead>
<tr>
<th></th>
<th>$C_1$</th>
<th>$C_2$</th>
<th>$C_3$</th>
<th>$C_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
<td>0.2</td>
<td>0.25</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>$C_2$</td>
<td>0.66</td>
<td>0.6</td>
<td>0.4</td>
<td></td>
</tr>
<tr>
<td>$C_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3. 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 are mainly C#, C++ and C. To identify evolutionary clusters we used historical information on file modifications, such as check-in meta-data, from the last nine years of development. We extracted the historical information from the ClearCase version management system.

When identifying evolutionary clusters we had to choose the level of abstraction for software entities. In one of our early discussions with the architects of the system, they told us they were primarily interested in investigating unwanted couplings at the level of building blocks. Building blocks are the next level of abstraction above individual files. A building block contains 50 files, on average. A building block is the unit of reuse. The internal structure of a building block is invisible to the other building blocks in the system. Usage of building block functionality is through import and export interfaces only; see also [19].

One abstraction level above building blocks we find subsystems. A subsystem is a set of building blocks mainly related to the same major functionality of the software system. The system comprises around 600 building blocks organized into 16 subsystems. The algorithm described in the previous section resulted in 595 evolutionary clusters involving building blocks from more than one subsystem.

In the software system studied, architects were not only interested in unwanted couplings that involved different subsystems. Sometimes, they were interested in unwanted couplings that involved different sets of subsystems. In particular, they were interested in unwanted couplings related to the following higher-level decompositions:

- development group decomposition,
- release group decomposition,
- deployment group decomposition, and
- development site decomposition.

The four decompositions above provide four different ways to group subsystems. These decompositions were defined by the architects of the system studied. Development groups are only allowed to modify the subsystems they are responsible for. At the moment of writing, the development group decomposition is not yet implemented in the organization. It is a decomposition which will be used in the future. In a mature situation, there would not be unwanted couplings between development groups, since different development groups would commit separately. In the present study, a development group decomposition is a fictitious one, used to assess a possible assignment of subsystems to development groups. 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. Depending on which decomposition of a software system we consider, the software entities of that system are partitioned into disjoint sets in different ways. In case it is not relevant which decomposition we are talking about we refer to those disjoint sets as decomposition elements in the sequel. In our study environment, a decomposition element can be, for instance, a single subsystem or the collection of subsystems of some given development group.

During the four years of our research (2006 till 2010) we interacted with three software architects and 27 software engineers. Interaction with these people happened in the form of interviews, questionnaires, surveys and workshops. In these four years, we developed a process to help the software architect to improve the decomposition of a software system, by approximating change sets [20], clustering change sets to determine evolutionary clusters, filtering unwanted clusters from the set of evolutionary clusters (the topic of the current paper) to the interactive analysis of unwanted clusters using visualization [14]. This process evolved over the years. We carried out three case studies, each lasting 6–8 months. In the first case study, we used a simple version of the step to select unwanted couplings. In the second case study, we extended this step to allow for the multidimensional characterization of evolutionary clusters discussed in the present paper. In the third case study, we elaborated the step to analyze unwanted couplings [14]. The lessons learned during this research are discussed in [21].

During the research presented in this paper we interacted with the lead architect to get feedback and validate our results. With him we had formal meetings nearly every week. Both the types of decomposition and the actual decompositions of the system we studied were given by the architect we worked with. That lead...
architect was actively involved in nearly every step of our research, including: the extraction of evolution anti-scenarios (Section 4), the identification and justification of evolutionary cluster properties (Section 5), and the translation of those scenarios to queries (Section 6). In a subsequent step the identified unwanted couplings had to be deeper analyzed, see [14]. During the latter analysis we interacted with the lead architect as well as the 27 developers.

As a result of the first case study, we were left with a large number of evolutionary clusters. Because the architect has limited time to handle issues, there arose the need to identify the most important of such clusters. To that end, we sorted the evolutionary clusters according to the frequency with which files in the cluster changed together, and presented the clusters whose elements changed most often to the architect. At a subsequent meeting, it turned out the architect had other criteria too to decide which clusters to investigate. For instance, clusters that involve more than one site were considered more important than clusters involving a single site. We next developed more complex metrics to use in the clustering algorithm, such as metrics that combine change frequency and the number of sites involved. This was however rejected by the architect, because he thought it would become too difficult to interpret the results of the clustering. As a consequence, we decided to use multiple simple measures to classify evolutionary clusters.

A main concern of the architect we mostly worked with was to ensure that development groups at different sites could work as independently as possible. So we focused on that concern. At a subsequent meeting we learned that a main concern of another architect was to ensure that pieces of software associated with different parts of the hardware could be deployed independently. So architect's concerns vary.

The combination of those two observations led us to devise the approach discussed here: the use of multiple simple dimensions to characterize evolutionary clusters, and the use of queries to filter out the unwanted clusters a particular architect is interested in.

Using Fig. 1, we can summarize how characterizing evolutionary clusters properly can help us to achieve the needs of the architects. We start with a large set of evolutionary clusters as represented by the biggest rectangle on the left. As described, different architects may have different responsibilities and therefore they are interested in different subsets of those evolutionary clusters. In Fig. 1, these different interests are denoted by the small rectangles named X, Y and Z. Personalizing the queries on the evolutionary clusters characterized according to the needs of different architects can help us to retrieve those different subsets. A sharper characterization allows us to better express which evolutionary clusters a software architect wants to analyze. Suppose an architect really is only interested in subset Y, but its superset X is retrieved using some characterization. In that case, the characterization leads to the analysis of a number of uninteresting clusters, the ones in \( X \setminus Y = \{ x | x \in [1 \cdots 6] \} \). Our experience in our study environment indicates that architects are more interested in reducing the time needed to analyze uninteresting clusters than finding all the clusters of interest. That is, it is more important to avoid false positives than to prevent false negatives. If we succeed in achieving that, we know the architect will only get a list of unwanted couplings he is interested in, albeit possibly an incomplete list.

4. Evolution anti-scenarios

As described above, what is considered to be an unwanted coupling depends on the architects or developers and their responsibilities. Once we know which dependencies experts are looking for, we are able to query the set of evolutionary clusters. Architects however are not thinking explicitly in terms of properties of evolutionary clusters when unwanted couplings are described. In our experience, architects tend to express what they consider unwanted couplings in terms of story-like 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 [22]. The next paragraph describes one such scenario.

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

This scenario is an example of what we call an evolution anti-scenario, analogous to the notion of design anti-patterns [23]. 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 unwanted couplings 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 coupling the architects and developers consider to be unwanted. Understanding which couplings are unwanted is the result of an iterative approach rather than the result of a one shot activity.

5. Properties of evolutionary clusters

Characterization of evolutionary clusters involves identifying the properties of those clusters and measuring the actual values for those properties. In this section we elaborate on the properties we use to characterize evolutionary clusters. In Section 5.9 we use one real-life evolutionary cluster from our study environment to illustrate different circumstances under which such an evolutionary cluster may be judged an unwanted coupling.
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 properties of the characterization. They should allow us to query evolutionary clusters and identify the types of unwanted couplings described by the evolution anti-scenarios. For every property we argue why it should be included in the characterization. We use actual evolution anti-scenarios from the study environment to support our arguments.

5.1. Property 1: Cluster size

The size of an evolutionary cluster is the number of building blocks involved. The size therefore may be an indication of its changeability. The bigger the size, the more building blocks need to be changed together and the more complex the whole operation might be.

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 building blocks which he thinks are affected by the change. Being new and having a lack of domain knowledge, however, \( D \) forgets to modify another ten building blocks from different decomposition elements. 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 building blocks 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 is therefore less complex.

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 building blocks less dependent, for instance by moving building blocks to another decomposition element.

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 unwanted couplings as possible from a fixed budget, then evolutionary clusters which indicate cheap-to-resolve unwanted couplings are probably the target for the resolution activities. The effort to resolve an unwanted coupling is likely to be influenced by the size of the evolutionary cluster. The bigger the evolutionary cluster is, the more building blocks are related and the more effort it may take to resolve the unwanted coupling.

5.2. Property 2: Borders crossed

Every building block in the evolutionary cluster can be identified using different decompositions. 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.

Let us reconsider the evolution anti-scenario described in Section 4. In that scenario, building blocks from development sites \( G_A \) and \( G_B \) had to be changed together. On average, it takes a lot more time to realize these non-local changes. If such non-local changes occur frequently, one may decide to move a complete subsystem to another site, or relocate part of one subsystem to another subsystem at a different site, to reduce site dependencies.

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

Typically, all of these various border-crossing relationships are important. It is a multi-dimensional separation of concerns and a careful weighing of concerns is required to resolve the associated unwanted couplings.

5.3. Property 3: Support count

The support count between two building blocks is the number of times those building blocks changed together, see [24]. This property indicates the distribution of the support counts for all the building block pairs \( \{BB_a, BB_b\} \) in the cluster where \( BB_a \) and \( BB_b \) are from different decomposition elements.

Consider the following scenario. In a software company a new project was started a year ago, to develop new and improved features for their software product. This necessitated changes to most of their subsystems. The realization of one feature in particular posed problems, and many iterations were needed before a satisfactory solution was obtained. At the end of the year, we observed that one set of building blocks \( X \) from subsystem \( S_X \) had changed together with a set of building blocks \( Y \) from subsystem \( S_Y \) 200 times. With this number of co-changes, \( X \) and \( Y \) are the building block sets which changed together most often in the last year.

In the above scenario the co-change of \( X \) and \( Y \) may indicate an unwanted coupling between those subsystems. Even if changing those sets of building blocks together is not the most costly operation, changing them together 200 times makes the underlying coupling a costly one. In case development activities will touch upon the same subsystems the next year, architects may want to resolve unwanted couplings similar to the one indicated by the co-change of \( X \) and \( Y \). Usually, architects are interested in the outliers when considering the support distribution.

5.4. Property 4: Jaccard similarity

The Jaccard similarity expresses the probability of two building blocks changing together given that one of them gets changed, see also [16,17]. The Jaccard similarity is a symmetric measure and it is a combination of the asymmetric confidence measures [24] (number of common changes divided by the number of changes to one of the building blocks). This property measures the Jaccard similarities of building block pairs from different decomposition elements.

Let us consider the following scenario. The architect decides that subsystem \( S \) needs to be outsourced, and his decision is approved by management. The decision is taken in order to reduce development costs. Company \( C \) takes over the development of \( S \). Although \( S \) is outsourced, it is still actively connected to the rest of the system, in particular to subsystem \( Y \). As a consequence, a change in \( S \) may require a change in \( Y \), and vice versa. After a year of co-operation it turns out that this dependency is indeed a strong one: in almost 90% of the cases in which a change in either \( S \) or \( Y \) was made, a change in the other subsystem was needed as well.
The communication and the increased management costs due to the poor isolation of $S$ from $Y$ exceeded the original development costs. Consequently, a decision is taken to in-source $S$ again.

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

5.5. Property 5: First co-changes

We can measure how many times building blocks changed together, but we can also observe when exactly they changed together for the first time. The date of the first co-change helps to understand since when the participating building blocks are related. This property captures, for a given system decomposition, the distribution of the first co-changes between building blocks from different decomposition elements. The first co-changes are indicated with the letter $F$ in Fig. 2.

Consider the following situation. Building blocks $A$ and $B$ often have changed together (i.e. their Jaccard similarity is high), and the same holds for building blocks $C$ and $D$. The dependency between $A$ and $B$ first arose a month ago, while the dependency between $C$ and $D$ has been there since the very beginning. It turns out that the dependency between $A$ and $B$ is caused by the introduction of a new feature which involved changes to both these building blocks. We expect that this dependency is temporary and will become less over time as the implementation of the feature becomes stable. On the other hand, the dependency between $C$ and $D$ denotes a recurring problem, and it is decided that it needs immediate attention.

In the software system we observed, it often takes a while after the introduction of a new feature before the concerns are well separated. Co-changes during that initial period need not indicate an unwanted coupling.

5.6. Property 6: Last co-changes

Similar to Property 5, this property expresses, for a given system decomposition, the distribution of the last co-changes of building blocks from different decomposition elements. The last co-changes are indicated with the letter $L$ in Fig. 2.

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

5.7. Property 7: Co-change tendencies

Co-changes between two building blocks have a certain distribution over time. Fig. 2 (a), (b) and (c) illustrate three such distributions. In each of the three cases thicker vertical line segments indicate co-changes between a pair of building blocks. The horizontal position of those segments is determined by when the co-change happened between the first co-change and NOW (the end of the time period analyzed). As indicated by Fig. 2, the distribution of co-changes may show different tendencies:

1. Co-changes getting more frequent (Fig. 2a).
2. Co-changes getting less frequent (Fig. 2b).
3. Co-changes having a more or less stable frequency of occurrence over time (Fig. 2c).

Similar to the Yesterday’s Weather approach of Girba et al. [25], we consider the tendency of co-changes to get a better understanding of what may happen in the near future. More specifically, tendency observations may help us understand which unwanted couplings are more likely to suffer from in the near future. To decide which of the described tendency types can be found in an evolutionary cluster we need to do some further measurements.

We determine the tendency type for two building blocks by mapping the period between the first co-change and NOW on a $[-1,1]$ interval, see Fig. 2. Then we determine the position of every co-change in this interval resulting in numbers $t_1, t_2, \ldots, t_N$ where $\forall i \in [1 \ldots N]$; $t_i \in [-1,1]$ and $N$ is the number of co-changes between the two building blocks analyzed. The average of all the resulting numbers $(\frac{t_1 + \cdots + t_N}{N})$ is referred to as the tendency number. A tendency number near to $-1$ tells us that the co-change frequency decreased, a tendency number near 1 shows that co-changes are getting more frequent over time and a value near 0 indicates that co-changes are evenly distributed over time. There may be more sophisticated ways to measure the tendency of co-changes than how we do it, but we picked our solution because of its simplicity. This property describes, for a given system decomposition, the distribution of the tendency numbers between building blocks from different decomposition elements.

Building blocks which changed together a lot but the frequency of co-change has become very low may not be interesting to the architects because at present those building blocks are less likely to change together. On the other hand, building blocks which have been changed 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-change frequencies between building blocks.

5.8. Property 8: Static relationships

With this property we count, for a given system decomposition, how many building block pairs from different decomposition elements are also coupled in terms of static relationships. The static relationships we considered in our study environment are include and call relations. These relationships can help to identify couplings between building blocks by just having a look at the content of the files in those building blocks.

Consider the following scenario. Suppose building block $A$ has changed frequently together with building block $B$ (so their Jaccard similarity is high). After a source code analysis it turns out that the co-changes of building blocks $A$ and $B$ are caused by their static coupling. The architect decides that this coupling has to be removed. There are two other building blocks $C$ and $D$ which also changed frequently together but without any static couplings between the two. Further analysis of $C$ and $D$ is not carried out since it is thought to be costly and time consuming. Building blocks $C$ and $D$ keep changing together until the additional development, testing and communication costs are so high that the co-changing $C$ and $D$ are further analyzed. The source code participating in the co-changes of $C$ and $D$ is identified and interpreted which reveals that $C$ and $D$ are semantically coupled. At the end, the costs of the coupling between $C$ and $D$ is higher than the costs of the coupling between $A$ and $B$. This is mainly caused by the fact that the coupling between $C$ and $D$ remained unresolved for a longer time.
The evolution anti-scenario just described teaches us that there are situations where evolutionary clusters that exhibit a low number of static relationships crossing the borders of decomposition elements are still important. One of the reasons is that an evolutionary cluster where the related unwanted coupling is caused by static relationships is likely to be already known and therefore its investigation may bring no additional information. Another reason is about the costs of unwanted couplings caused by non-static relationships. The experts we talked to agreed that around 10% of the unwanted couplings are not caused by static relationships. Those are the unwanted couplings which remain unknown for a longer time because of the difficulty to identify them and therefore they are considered to be more harmful, i.e. costly, than other unwanted couplings.

5.9. An example

Fig. 3 illustrates one evolutionary cluster characterized in the context of the release group decomposition in our study environment. The outer rectangle represents the software system. The solid lines illustrate the release group decomposition of the software system. Dashed lines further refine that decomposition by indicating the borders of subsystems. Recall that a release group is a set of subsystems. The small circles indicate those building blocks which are part of the evolutionary cluster characterized. So the evolutionary cluster contains seven building blocks from two subsystems. These subsystems are in different release groups. A line connecting two building blocks (from different subsystems) indicates these building blocks were changed together; the number in the black rectangle indicates how often they were changed together.

An evolutionary cluster such as the one depicted in Fig. 3 is the result of (in general) many individual change sets. Not all of these change sets need to be identical (as evidenced by the different numbers in the black triangles in Fig. 3). Some may involve more building blocks from each participating subsystem, others less. For instance, one change set may consist of building blocks \{BB_2, BB_3, BB_7\}, while another one just contains building block BB_5, and a third one contains all seven building blocks. For many properties, we therefore consider the maximum, minimum, average and the standard deviation of the property values over all change sets participating in the evolutionary cluster (as in Table 4). Since the number of building blocks in the set of change sets that defines an evolutionary cluster is from a relatively small range, using the average is meaningful. Expressing the properties measured in terms of their minimum, maximum and average values is important to quickly select those evolutionary clusters architects are interested in. For instance, if we want to select evolutionary clusters whose property value is larger than some value $X$, knowing the minimum, maximum and average helps to make a good guess about an appropriate value for $X$.

For the various properties introduced above, the values for the evolutionary cluster from Fig. 3 are as follows:

- **Cluster size** The evolutionary cluster contains seven building blocks (BB_1–BB_7).
- **Borders crossed** The 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 well. As for the other decompositions, the cluster is located in a single decomposition element.
- **Support count** The evolutionary cluster contains release group crossing relationships with a relatively high support, see Table 4. Building blocks from different release groups in this evolutionary cluster were changed from 32 to 107 times, and on average 66 times.
- **Jaccard similarity** For the release group decomposition of our example, the maximum, minimum, average and the standard deviation of the Jaccard similarity values are depicted in Table 4.
- **First co-changes** The data in Table 4 shows the maximum, minimum, average and the standard deviation (given in days) of the first co-changes. As we can see from Table 4, all building blocks participating in the analyzed evolutionary cluster were first changed together on the same day. This holds for the evolutionary cluster observed; it is most often, however (in 70% of the evolutionary clusters analyzed in our study) not the case.
- **Last co-changes** Table 4 shows the distribution values for the last co-changes.
often difficult to a priori decide on a proper value. By trying different properties, such as the Jaccard similarity and co-change tendency, it is easy to see if the query has been too strict, or not strict enough. For some properties, we usually start with the cluster size and borders crossed properties, since these are easiest to grasp. The properties used for the characterization and the effects of using different values, and based on the knowledge the architect has of the system, an acceptable output is obtained. Such happened for example in the case discussed in Section 6.1 where the first iteration resulted in a too large set of unwanted couplings, and the architect wanted to sharpen the criteria to get a smaller set.

In the next subsections, we describe two real-life cases when the elicited evolutionary scenario(s) were translated to queries. In Section 6.3 we discuss the gains of using anti-scenarios and the corresponding queries to filter out unwanted couplings in these cases.

### 6. Querying evolutionary clusters

In order to find evolutionary clusters which point to unwanted couplings we need to create and execute a query on the set of evolutionary clusters. Such a query has to result in evolutionary clusters, where the corresponding dependencies are considered unwanted by a specific architect or developer. To create the query, we use the evolution anti-scenarios elicited from the architect or developer we want to support. We translate the anti-scenarios to queries together with the software architect during formal meetings. During such a meeting an architect needs to get familiar with the properties used for the characterization and the effects of using those properties in the query. In our case study, it took us approximately one hour to agree on the precise formulation of the query. The process of creating the set of unwanted couplings thus comprises four steps:

1. elicit the anti-scenarios from the architect,
2. explain the properties to the architect,
3. translate the anti-scenarios into values for each property, and
4. execute the resulting query on the set of evolutionary clusters.

The anti-scenarios provide a starting point for the discussion of the various properties to be used (Step 2). When explaining the properties, we usually start with the cluster size and borders crossed properties, since these are easiest to grasp. The properties are explained in terms of the anti-scenarios, so that an understanding is built of their mutual relation. Steps 3 and 4 may have to be iterated several times before an acceptable result is obtained. After selection of a set of values and executing the query, one may inspect the resulting set of evolutionary clusters and decide whether the query has been too strict, or not strict enough. For some properties, such as the Jaccard similarity and co-change tendency, it is often difficult to a priori decide on a proper value. By trying different values, and based on the knowledge the architect has of the system, an acceptable output is obtained. Such happened for example in the case discussed in Section 6.1 where the first iteration resulted in a too large set of unwanted couplings, and the architect wanted to sharpen the criteria to get a smaller set.

In the next subsections, we describe two real-life cases when the elicited evolutionary scenario(s) were translated to queries. In Section 6.3 we discuss the gains of using anti-scenarios and the corresponding queries to filter out unwanted couplings in these cases.

### 6.1. Case 1: Development group unwanted couplings

In the first case, the architect had the task to make sure that the development groups depend as little as possible on each other. To identify the evolution anti-scenarios related to this case we organized a meeting with the architect. We started that meeting with asking the architect to recall which were so far the most severe development issues he had to face caused by unwanted couplings. The evolution anti-scenarios which we elicited from the architect in this case describes (1) a large amount of communication between different development groups, and (2) delay of work of some development groups because they have to wait for other development groups. One such scenario from the architect is the following:

“Once we have decided to outsource subsystem A to development site DS. We believed that the couplings between subsystem A and the rest of the system is low enough to make the outsourcing successful. However, the amount of communication with development site DS increased drastically, overruling the benefits of outsourcing. This was a costly lesson to learn.”

In a second meeting with the software architect we created a query on the evolutionary clusters. Since this was the first such exercise for the architect, we started by explaining the properties we used for characterizing the evolutionary clusters, see Section 5. During the one-by-one explanation of the properties we made sure the architect really understood what the properties were about. Next, we discussed which of the characterization properties had to be used in the query, and which thresholds to use.

First we discussed the cluster size property. If more than one development group is involved in a change, then the number of building blocks involved, and therefore the complexity of the change, does have an influence on how much communication effort is spent. Furthermore, more building blocks typically take more time to modify and in such cases development groups may delay other development groups for a longer time. Therefore we chose to include the size of the evolutionary clusters in the query.

Next we looked at the border crossing properties of evolutionary clusters. It was straightforward to decide to select the evolutionary clusters crossing the borders of development groups, since the architect was interested in the unwanted couplings of the proposed development group structure and not in the unwanted couplings of other structures.

We continued with a discussion of the support property. The support property was included in the query because we wanted to identify unwanted couplings that indicate frequent communication between development groups. The more frequent develop-

### Table 4

Property values of the example evolutionary cluster.

<table>
<thead>
<tr>
<th>Property</th>
<th>MAX</th>
<th>MIN</th>
<th>AVG</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support count</td>
<td>107</td>
<td>32</td>
<td>66</td>
<td>21</td>
</tr>
<tr>
<td>Jaccard similarity</td>
<td>24.71%</td>
<td>8.29%</td>
<td>15.42%</td>
<td>4.52%</td>
</tr>
<tr>
<td>First co-change</td>
<td>15 July 2004</td>
<td>15 July 2004</td>
<td>15 July 2004</td>
<td>0 days</td>
</tr>
<tr>
<td>Last co-change</td>
<td>30 April 2009</td>
<td>19 August 2008</td>
<td>28 January 2009</td>
<td>104 days</td>
</tr>
<tr>
<td>Co-change tendency</td>
<td>−0.16</td>
<td>−0.30</td>
<td>−0.23</td>
<td>0.04</td>
</tr>
</tbody>
</table>
ment groups need to communicate, the more effort on communication needs to be spent.

The Jaccard similarity was the next property we discussed. Evolutionary clusters with a low average Jaccard similarity are a consequence of many changes that are local to a development group, and few changes that involve more than one development group. Such evolutionary clusters therefore denote the correctness of the development group decomposition rather than an unwanted coupling. Consequently, we decided to filter out evolutionary clusters having a low Jaccard similarity average.

Next we looked at the first and last co-change and co-change tendency properties. We explained the architect that if building blocks last changed together a long time ago, e.g., the last co-change between those building blocks happened 5 years ago, then this probably indicates an already solved issue. We also explained to the architect that even if the first co-changes of building blocks happened relatively recently (a few weeks ago), the frequent co-changes of those building blocks would not provide us with enough historical evidence to say that we are facing an unwanted coupling. The architect agreed that using the first co-change property in the query is indeed helpful.

The architect was very positive about the idea of using the co-change tendency property. According to him, knowing if the co-changes are getting stronger or weaker is important to decide which unwanted couplings are more severe than others. When explaining the tendency property to the architect, we showed him a few examples of building blocks changing together over time. After looking at those examples the architect expressed that he is most interested in those unwanted couplings where the co-changes of the participating building blocks occur periodically. This translates to a tendency number close to zero. The architect also argued that those cases where the co-changes got more frequent, i.e. the tendency number is positive, are less relevant since the frequent co-changes are only recently observed.

After having discussed all properties, and having decided which ones to include, we set the threshold values for the properties selected. To express the interest of the architect setting one threshold per property was enough. This again involved several iterations. We started with an excel sheet containing all clusters that had a Jaccard similarity level >0.05. This excel sheet contained 178 of the 595 evolutionary clusters. The columns of the sheet contained the minimum, maximum, average and standard deviation for all properties. Browsing this sheet gave the architect a feeling for the distribution of the various properties. For example, the size of the clusters ranged from 2 to 42. Based on these ranges, the architect next decided upon the thresholds to be used. The order in which the thresholds were decided upon reflect their relative importance to the architect. For instance, he considered the co-change tendency more important than the first co-change numbers. In the case of co-change tendency the architect first opted for the value 0.2. This is a relatively high tendency number which also expresses his need: to find unwanted couplings where the co-changes are occurring periodically. After executing the query with that tendency number threshold and analyzing the sheet with the resulting evolutionary clusters, we found however that the threshold set was too strict: the architect also wanted to find unwanted couplings with a somewhat lower threshold. Therefore we modified the threshold to –0.15 and reran the query. The resulting sheet contained some additional evolutionary clusters pointing to couplings the architect deemed unwanted. This example shows that multiple runs of the query are needed to get a good feeling about which thresholds to use. The final query on the evolutionary clusters is shown in Table 5.

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 unwanted couplings independently from whether they are easy to identify from the static relationships or not.

### 6.2. Case 2: Release group unwanted couplings

In the second case we consider an architect who had the task to make release groups more independent. The final goal was to release new versions of every release group on their own, without mutual dependencies. As a first step, static relationships were analyzed to know at which points release groups are coupled. This step was thought to be relatively cheap and effective, but looking only at static relationships did not allow the architect to identify all the unwanted couplings. An additional argument was that co-changes without static dependencies are much more costly than the ones caused by static dependencies. Therefore the architect was interested in couplings where the unwanted co-changes were not caused by direct static dependencies. This can be the case for instance if two building blocks implement one and the same data model. This then introduces a semantic dependency between those building blocks which is likely to show up in changes involving that data model. The evolution anti-scenario in this case describes that building blocks from different release groups cannot be released independently although static dependencies are removed. Based on this scenario we created the query on the evolutionary clusters with the criteria described in Table 6.

We determined the thresholds for the characterization properties very similar to how we have done it in the previous case. First, we considered only the evolutionary clusters where there were no static relationships between the building blocks included. Second, the release groups were of interest rather than the development groups. 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 for these lower thresholds is that the developers working on the system analyzed know that there are far fewer unwanted couplings caused by non-static dependencies. It is important to the architect to know of evolutionary clusters even if the building blocks changed less frequently together or the cluster contains fewer building blocks.

### 6.3. Comparing selection approaches

How much did we actually gain by using a more sophisticated selection of evolutionary clusters, as compared to the selection...
mechanism suggested by previous work? In doing so, we have to keep in mind that false positives are more harmful than false negatives. Analyzing whether an evolutionary cluster is an unwanted coupling or not takes a considerable amount of time (a couple of hours at least), even if we reject the cluster at the end. Our experience is that an architect has a very limited time and cannot spend weeks on a single task. Analyzing, for instance, all evolutionary clusters we identified would have required too much time from the architect.

To answer the above question we measure how many evolutionary clusters previous approaches and our approach sort out from the complete set of evolutionary clusters identified. The better we can retrieve the evolutionary clusters which the supported architect acknowledges to point to unwanted couplings, the better our approach is. We use Case 1 described in Section 6.1 during the comparison.

Altogether, we identified 595 evolutionary clusters. A vast majority of the related work dealing with the identification of unwanted couplings, see [13,12,26,10], suggests that software entities (such as building blocks in our case) from different subsystems may point to an unwanted coupling if those entities changed together frequently. Using the terms of our characterization it means that the interesting evolutionary clusters are crossing the borders between subsystems (or modules) and that the co-change relationships between entities from different subsystems have a high support. We do not consider commonly used criteria (support and support properties to filter co-changings) as Default. If the architect had specified only those commonly used filtering criteria, 68 from a total of 595 evolutionary clusters would have been selected.

Some earlier work extends the common way to find entities participating in unwanted couplings. Next to the subsystem crossing and support properties, German [8] and Zimmerman et al. [24] consider the confidence between entities as well, very similar to what we do with the Jaccard measure. Recall that the Jaccard measure (Section 5.4) is a combination of confidence measures. We refer to the criteria used here as Default + Jaccard. Applying this additional filter results in 25 evolutionary clusters. Antoniol et al. [4] are using the size of the evolutionary clusters together with the support and subsystem crossing properties to filter co-changings and find unwanted couplings. The criteria used by Antoniol et al. is further referenced as Default + Size. In Case 1, querying evolutionary clusters based on these three properties results in 36 evolutionary clusters.

In Case 1 we needed seven selection criteria to express which evolutionary clusters the architect was interested in (see Table 5). Using these seven criteria in the query results in 12 evolutionary clusters. We next used the interactive visualization approach to further analyze with the architect and developers the topmost 10 of those 12 evolutionary clusters selected, see also [14]. Each of these clusters points to an unwanted coupling as acknowledged by the developers. Compared to the default way to find unwanted couplings, we did not select 56 evolutionary clusters (82%). The architect was not interested in these clusters because, for instance, they were likely to point to an already solved dependency (when, e.g., the last co-change happened more than 2 years ago) or too little evidence has been gathered about the co-changing files (when, e.g., the first co-change happened less than 2 months ago). Similar to Case 1, we executed the query in Case 2 as well. That query also results in a small number of evolutionary clusters, namely ten.

By querying the set of evolutionary clusters we identify a subset thereof. Analyzing the selected subset, however, is typically a sequential process where those clusters are analyzed one after the other. Furthermore, an architect may have a preference to analyze some evolutionary clusters sooner than others. This implies that evolutionary clusters need to be prioritized when we present them to software architects. Many of the related work suggest to base this prioritization on the number of co-changes, i.e. on the support count. Also, the architect we worked with wanted to analyze the evolutionary clusters in decreasing order of their support count. Such a sorting makes it possible to compare approaches selecting evolutionary clusters.

The criteria we used in Cases 1 and 2 to query evolutionary clusters form a superset of the criteria used by previous work. Therefore, the evolutionary clusters we selected in Cases 1 and 2 form a subset of the evolutionary clusters selected with the criteria of the previous work. We can then observe where the evolutionary clusters selected in Cases 1 and 2 end up in the ordered lists of evolutionary clusters we get using the selection criteria of the previous work. Table 7 and 8 show the results of the comparisons. During the comparison we used the three groups of related works mentioned, where the criteria to filter evolutionary clusters are: Default, Default + Size and Default + Jaccard.

From Table 7 we can see, for instance, that the third evolutionary cluster which the architect analyzed in Case 1 would have been the ninth cluster analyzed using the default criteria for querying the clusters. The architect who was interested in the evolutionary clusters that result from the query in Case 1 should have analyzed 34–4 = 30 clusters, queried using the default criteria, before he would have found the fourth cluster he was actually interested in. That is, there are 30 false positives according to the scenario used in Case 1. We can also see that using the size and Jaccard similarity properties in the query further helps to focus on relevant clusters but the number of false positives can still be as large as 17–4 = 13 (with Default + Size) or 12–4 = 8 (with Default + Jaccard). Similar results hold for Case 2, see Table 8, as well.

Note this comparison is not intended as a critique on the research described in papers such as [13,24,4]. That research was not intended to support architects dealing with unwanted couplings. The argument we put forth is that it is useful to look at other dimensions as well.

7. What if we move some files?

In Cases 1 and 2, discussed in Sections 6.1 and 6.2, the set of evolutionary clusters was queried to identify unwanted couplings. Knowing the unwanted couplings, however, is only the starting
point for architects to improve the structure of the software system. After identifying an unwanted coupling an architect needs to:

- understand why the unwanted coupling exists,
- identify the alternatives to solve or mitigate the unwanted coupling,
- decide upon a solution to be implemented, and
- carry out the structural changes suggested by the solution chosen.

Querying the set of evolutionary clusters can be used beyond the identification of unwanted couplings. It may also help in the subsequent process when software architects have to decide upon an alternative solution. This is an important lesson we learned when we, together with the software architect and several developers, applied a what-if type of analysis on one unwanted coupling that resulted from executing the query from Case 1 (Section 6.1).

The what-if type of analysis can be done if a solution alternative proposed by the experts is about moving some software entities, i.e. files, building blocks, or subsystems, from one decomposition element to another. In our case, such alternative solutions were suggested for half of the evolutionary clusters analyzed with the experts. An analysis thereof is necessary if the software architect is not sure about the consequences of implementing an alternative solution for an unwanted coupling. Our what-if type of analysis consists of the following steps:

- Alter the paths of the software entities which are to be relocated. The paths have to be changed in the versioning metadata used to create the evolutionary clusters. As a result we have a historical data set which mimics the situation where the software entities to be relocated reside in their new decomposition element.
- Re-run the approach used to identify evolutionary clusters on the data set resulting from the previous step.
- Apply the same query on the newly identified evolutionary clusters.
- Analyze the difference between the original and newly queried evolutionary clusters. During such a comparison we are especially interested in knowing whether the software entities moved are participating in any of the newly identified evolutionary clusters (new unwanted couplings would be introduced). Also we need to know if the original evolutionary cluster analyzed is still present in some form in the new set of evolutionary clusters (i.e., the coupling is not (completely) solved).
- Presenting the results to the architect such that he can take them into account when deciding about the solution alternative in discussion.

In the remainder of this section we discuss the actual instance of the what-if type of analysis which we executed in our study environment for Case 1, see Section 6.1.

The unwanted coupling investigated involves C source files from one subsystem and C header files from another subsystem. The file level analysis revealed that source files had been changed together with header files very frequently (more than 20 times). As those subsystems were supposed to be maintained by different development groups, the frequent co-changes between the source and header files pointed out a costly and, therefore, unwanted, coupling.

For the file level analysis of the evolutionary cluster we used our interactive visualization tool iVis [14]. Fig. 4 shows a snapshot of the tool while it is visualizing the evolutionary cluster under discussion. The tool presents software entities (subsystems, building blocks, files) and the co-changes between those entities. The outer blobs in Fig. 4 represents the two subsystems the C source and header files are coming from. The tool was configured such that it only shows a line between files (represented by white squares) if they were changed together at least 20 times. As we can see, there are many strong relationships crossing the borders of subsystems and therefore contributing to the unwanted coupling. Further information on the tool can be found in [14].

Meetings organized with the architect and some of the developers involved in the development of the C files under discussion helped us identify the underlying reason for the unwanted coupling. Originally, both the source and the header files were designed to be in the same subsystem but a recent restructuring activity had put them apart. It has been acknowledged by the developers to be a mistake made during the restructuring.

The developers proposed a solution to the architect to solve the unwanted coupling. They suggested moving both the source and the header files under discussion to a third subsystem. The architect had doubts about the adequacy of the solution proposed. He expected that moving the source and header files involved in the unwanted coupling would give rise to new unwanted couplings related to those files. Therefore, the architect wanted to know if moving the files to their proposed location would indeed introduce new unwanted couplings and if the coupling discussed would be resolved. Knowing the answers to these questions was a key issue for the architect to decide upon the acceptance or rejection of the solution alternative proposed by the developers.

To help answer the questions of the architect we redefined the location of the source and header files under discussion as if they had been moved according to the suggestion of the developers, by changing the paths of the relevant files in our database containing the historical meta-data from ClearCase. Using the modified input we identified, characterized and queried the evolutionary clusters the same way we had done before in Case 1, see Section 6.1. By comparing the newly selected evolutionary clusters to the originally selected ones we made the following observations:

- None of the source and header files under discussion are part of the newly selected evolutionary clusters.
- None of the building blocks originally containing the discussed source and header files are part of the newly selected evolutionary clusters.
8. Lessons learned

This paper discusses one step in a process to help architects assess the decomposition of a system. The overall lessons learned in devising this process over a 4-year period are discussed in [21]. Here, we focus on the specific lessons learned while doing the multi-dimensional characterization of evolutionary clusters in our study environment:

- Selecting unwanted couplings using a richer set of properties significantly reduces the number of evolutionary clusters that need to be investigated. Tables 7 and 8 show the reduction in the number of false positives over our approach yields, compared to selection using only the support count and borders crossed properties. Since architects are under time-pressure, they can inspect a limited number of unwanted couplings only. The reduction obtained using a richer set of properties helps focus the architect’s attention to issues that really matter to him.

- Querying the set of evolutionary clusters cannot only be used to identify unwanted couplings an architect is interested in, but it also can be used to execute what-if scenarios, as exemplified by the example in Section 7. If the architect wants to assess the effect of a structural change, he can do so by changing paths in the versioning meta-data, generate a dataset which mimics this new situation, and run the approach using the new dataset. This what-if type of analysis helps to quickly assess the suitability of a structural change before it is actually being implemented.

- Formulating the query to be applied involves an iterative process wherein the criteria to be used gradually develop. Architects have to build insight into what issues they consider worth addressing. The use of evolution anti-scenarios – story-like descriptions of situations to be avoided – provides guidance in determining which properties to use, and what their values should be.

- Concerns of architects vary. Different concerns lead to different queries on the set of evolutionary clusters. Case 1 and Case 2 described in Sections 6.1 and 6.2 show that different queries and the resulting set of unwanted couplings can be very different. This finding shows once more that the notion of an unwanted coupling is subjective.

9. Threats to Validity

In this section, we list possible limitations to our experience report by discussing the internal and external validity, following [27,28]. Internal validity relates to the extent to which the results of our case studies may have been biased by confounding variables and other sources of bias. External validity relates to the extent to which any conclusions can be generalized to settings outside that of the current study.

9.1. Internal validity

With respect to internal validity, we discuss the following threats:

- we had no complete freedom in the selection of cases to explore,
- we used the opinion of one architect only,
- the results may be sensitive to the threshold values used, and
- the accuracy of the evolutionary clusters identified depends on the accuracy of the identified co-changes.

Our study is not a controlled experiment. In particular, we were not able to pick a random set of anti-scenarios and corresponding queries, and evaluate the resulting collection of unwanted couplings. We had to use the anti-scenarios that had top priority for the architect. Only then could we involve him in our study. In [29], David Harel reflects on his early work on Statecharts, and observed: “One of the most interesting aspects of this story is the fact that the work was not done in an academic tower, inventing something and trying to push it down the throats of real-world engineers. It was done by going into the lion’s den, working with the people in industry. […] If what you come up with does not jibe with how they think, they will not use it. It’s that simple.” Our approach is similar. We worked together with a real architect, on real issues he is concerned about, and describe our experiences therewith.

Based on the distributions presented, the architect determined the property thresholds we applied during the query of the evolutionary clusters. Adjusting the properties slightly may result in a different set of evolutionary clusters being selected. In future work, we need to do further studies to learn about the sensitivity of the different property thresholds.

The accuracy of identifying co-changes between software entities may have an effect on the measured evolutionary cluster properties. In an extreme case, a single wrongly identified co-change may have a direct effect on the MAX and MIN measures. To maximize the accuracy of the evolutionary cluster characterization, we have to make sure that the co-changes are identified as accurately as possible. The latter we studied separately [20].

9.2. External validity

With respect to external validity, we discuss the following threats:

- we worked with one architect only,
- we pay attention to precision of results only, and not recall, and
- we applied the approach in one environment only.

In order to create the queries on the evolutionary clusters characterized we had meetings with one architect only. During those meetings we have also learned that other architects may consider the severity of couplings differently, because of their different responsibilities and/or actual interests. This is also reflected in the differences between Case 1 (Section 6.1) and Case 2 (Section 6.2). In order to present more solid benefits of our characterization approach we should contact more architects. However, the architect we worked with did have a complete overview of the software systems’ architecture and its issues. This is because the architects of the software system analyzed closely work together and share their concerns.
In assessing the accuracy of the subset of evolutionary clusters marked as unwanted couplings, we consider precision only. In our study environment, architects are under time pressure, and want to spend their limited time on investigating real issues only. That is, we need to prevent false positives. The queries we formulate may lead to false negatives, i.e., we may fail to identify some of the unwanted couplings. Checking for false negatives was not possible for us because, for instance, analyzing all the 68 evolutionary clusters in Case 1 would have taken too much effort from the architect. But since the architects only have time to investigate so many of them, this was less of a concern. The fact that our characterizations did not result in any false positives helped to convince the architect of the value of our approach.

We have executed our approach in one study environment only. For example, it is not guaranteed that in every environment the initial set of properties as defined in Section 5 is complete enough to express what architects consider to be an unwanted coupling. 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. To be able to further generalize our results, experiments with our approach in different environments are needed. However, our results from this specific, highly competitive, industrial environment are promising. Also, we have assumed relatively little about our study environment. Below, we elaborate for every step of our approach which are the major points of concern one may need to think about when executing our approach in a different environment.

Our approach works on multiple input sources:

1. the meta-data on changes extracted from the version management system,
2. the evolutionary clusters containing software entities which frequently changed together,
3. the decompositions of the software system to be evaluated and improved, and
4. the abstraction level of interest for the software entities in the evolutionary clusters.

The change meta-data we used from a version management system about who changed what and when is widely available. This is even true in those environments where early version management systems, such as CVS, are being used. Based on this generally available input the evolutionary clusters can be constructed, for instance, with the approach we described in [5].

Our approach is used to identify unwanted couplings in one or more decompositions of the software system. The approach assumes that the decompositions to be assessed are known. We only assume that the decompositions partition the software entities of the system into decomposition elements. In that sense our approach works with any kind of system decomposition.

The abstraction level of interest for the software entities in the evolutionary clusters, e.g. building blocks in our study environment, can be anything as long as we can observe when the entities changed together. Selecting a low abstraction level only poses computational challenges on the evolutionary cluster identification because of the typically huge number of software entities.

After acquiring the input data the next step is to characterize the evolutionary clusters. This step can be done in an automatic way, using an algorithm. Therefore, executing this step is straightforward and applicable in any environment where the previously described input data is available.

Querying evolutionary clusters is the last step to characterize the evolutionary clusters. How this step is executed may vary depending on the environment we are considering. The major point here is that one needs to contact the architect (or someone else in the organization) knowledgeable about the evolution anti-scenarios and with whom one can translate those scenarios to queries on the evolutionary clusters.

10. Related work

Finding unwanted couplings in a software system is addressed in many studies [5,26,30,10,31]. The structure of a software system has to fulfill a set of requirements in order to support the development and maintenance activities. For instance, semantically related files should be in the same subsystem to help developers achieve consistent modifications. Another requirement often posed on the structure of the software system is that static relations should exist between files from the same subsystem rather than between files coming from different subsystems. Depending on which requirement posed on the system’s structure is addressed, related work looking for unwanted couplings can be categorized. For our current work, studies which assess the independent evolution of decomposition elements, such as subsystems, are relevant. We used their results as an inspiration to characterize and query evolutionary clusters.

Antoniol et al. [4] identify groups of co-changing files in order to find non-trivial dependencies between decomposition elements. Such dependencies are expected to denote unwanted couplings in the software system. When characterizing the groups of co-changing files, Antoniol et al. consider (1) the size of the groups (i.e. the number of files included), (2) how many times files in the groups changed together and (3) if the files from the same group are coming from different decomposition elements of the software system.

Gall et al. [3,7] detect logical couplings between software entities, i.e. subsystems, modules and programs, by comparing in which sequences of releases they have been changed. The logical coupling between two entities is then measured by the number of releases where both entities got changed. The logical coupling between subsystems should be low to achieve a good modularization. The more two subsystems are logically coupled the higher the chance is that we are facing an unwanted coupling.

Zimmerman et al. [24] define coupling and cohesion measures to assess the modularity of software systems. To achieve their goal, they measure (1) how many times pairs of lower level program entities, such as functions, methods and attributes were changed (checked-in) by the same developer and (2) what the ratio is of the number of common changes and the total number of changes, i.e. they measure confidence values. If a pair of software entities coming from different decomposition elements of the system were changed together frequently, and with high confidence values, then it negatively impacts the level of modularity. The Jaccard similarity value we measure is in fact a combination of the confidence values Zimmerman et al. use.

German [8] characterizes different types of modification requests (MRs), which are approximated change sets. Our work takes the characterization further to the level of evolutionary clusters which are identified based on the change sets. Furthermore, German uses MRs containing source code files to derive modification coupling between pairs of files. For each file pair he measures (1) in how many MRs both files were present, i.e. co-changed and (2) confidence values, which is the likelihood that those files will be changed given that one gets changed.

Breu and Zimmermann [32] 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.
Mulder and Zaidman [33] mine frequent itemsets from software repositories to identify cross cutting concerns. Frequent itemsets are sets of items (in our case building blocks) that frequently change together. An itemset thus is a subset of a change set. Determining frequent itemsets does not involve the type of hierarchical clustering we use.

Using visualizations is an alternative way to identify unwanted couplings. When using a visual approach, it is often up to the intuition of the user of the visualization to find and scope the evolutionary clusters. In that case, identifying evolutionary clusters is somewhat subjective.

The visualization of Ratzinger et al. [26] helps the user find unwanted couplings by creating a visual overview on the classes which frequently changed together from different modules. D’Ambros [12] et al. define a layout of files to see which one of them changed together with a selected module of interest. Typically, one looks for unwanted couplings by identifying patterns in the visualization. Which pattern instance is selected to be further analyzed depends on one’s intuition. Pinzger et al. [34] help the formulation of such an intuition by visualizing multiple evolution metrics for several releases of the software system.

The visualization of Beyer and Hassan [10] provides a layout for software entities such that software entities which change together more frequently in the past are placed closer together. When those software entities are colored according to the decomposition element they are coming from then it is possible to observe the differences between the decomposition of the software and the software entities frequently co-changing. Such differences may point to unwanted couplings. Furthermore, to show how unwanted couplings evolved over time, [10] propose a storyboard-like animated visualization.

Fisher and Gall [13] also identify unwanted couplings using a visualization. But before visualizing files and their relationships they first filter out the “interesting” file pairs, i.e. files from different decomposition elements which changed frequently together. These interesting file pairs form the input for a visualization to detect anti-patterns, such as god-classes. Although the visualization indicates structural anti-patterns, the approach is not addressing explicitly the prioritization of unwanted couplings.

Treude and Storey [35] 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.

By reviewing previous work we observe that co-changing software entities are characterized in a rather simple way. The characterization is typically based on the number of common changes (support) and on the observation whether the co-changed entities are coming from the same decomposition element. Furthermore, visualizations of co-changes are also mainly based on those two properties. As a consequence, the users of such visualizations have to find and prioritize unwanted couplings based on those few properties only. In Section 3 we argued that it is important and useful to consider a broader characterization.

11. Conclusion and future work

In this paper we describe how to characterize evolutionary clusters and how this characterization may help architects and developers define what constitutes an unwanted coupling in the structure of a software system, and next find those couplings. After a remedy to an unwanted coupling is proposed, we can redo the characterization to a decomposition that is expected to solve the dependency, and compare the outcomes. This ‘what-if’ type of analysis allows for a quick assessment of changes to the software archive. A similar type of analysis can be done when the organization changes, for example when a group of people moves to another site, or development groups get reorganized. The ‘what-if’ analysis will then identify possible unwanted couplings because of that organizational change.

We show that a well prepared characterization of what constitutes an unwanted coupling is important to support software architects and developers. We describe properties of evolutionary clusters relevant for the characterization. We found that, in practice, evolution anti-scenarios are an important source of information on what is considered to be an unwanted coupling. Since the characterization itself may evolve over time, it is important to retain it.

Characterizing evolutionary clusters and querying them are only the first steps towards the identification of unwanted couplings. In a subsequent step, 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), and (3) what is the reason that those software entities changed together. A good visualization supporting the exploration of evolutionary clusters is essential not only to further analyze them, but also to investigate what could cause an unwanted coupling. We address these issues in [14].

The lead architect we contacted frequently during the characterization was positive about the usefulness of the approach described in this paper. The architect expressed several times that he especially liked the possibility to query evolutionary clusters based on the co-change tendency property. For him, resolving those unwanted couplings which repetitively caused problems in the past are a first priority. Furthermore, thinking about characterization explicitly helped refining the abstract requirement of independent evolution and to express sharper what an unwanted coupling is.

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

This study was done in one environment only. Doing similar studies for other large systems is an obvious way to help generalize our findings. Next to using version management data only, it would be interesting to explore the use of other coupling data, such as run-time couplings, in assessing the decomposition of the system. Finally, it would be interesting to explore changes in the couplings over time, to support architects in maintaining a good decomposition of the system.

Acknowledgments

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
