Retrieving Task-Related Clusters from Change History

Martin P. Robillard and Barthélémy Dagenais
School of Computer Science
McGill University
Montréal, QC, Canada
{martin,bart}@cs.mcgill.ca

Abstract

During software maintenance tasks, developers often spend an important amount of effort investigating source code. This effort can be reduced if tools are available to help developers navigate the source code effectively. For this purpose, we propose to search the change history of a software system to identify clusters of program elements related to a task. We evaluated the feasibility of this idea with an extensive historical analysis of change data. Our study evaluated to what extent change sets approximating tasks could have benefited from knowledge about clusters of past changes. A study of 3 500 change sets for seven different systems and covering a cumulative time span of close to 12 years of development shows that less than 12% of the changes could have benefited from change clusters. We report on our observations on the factors that influence how we can use change clusters to guide program navigation.

1. Introduction

When involved in a task to change unfamiliar code, a software developer will generally spend an important fraction of the task time investigating the code. In many development environments, investigating the source code can be supported in a wide variety of ways, from the most basic cross-reference searches (e.g., for the callers of a method) to advanced tools that take advantage of ever-growing quantities and types of software development data [18]. Examples of advanced tools and techniques to support software investigation include query-based source code browsers [9], association rule mining of change history [20], searchable project memory [5], automated feature location [16], and topology analysis of software dependencies [12]. The rich and diverse collection of available program investigation tools and techniques is not surprising when we consider the wide variety of questions developers ask themselves during software change tasks [14]. In fact the increasing size of most software systems motivates the development of a collection of search tools that can maximize the efficiency of developers in different program investigation situations.

In this context, we investigated the usefulness of a software's revision history for facilitating software investigation. To do so, we devised a technique that takes as input a simple query, and determines if there exists any *change clusters* that would match the query. The concept of a change cluster has been used in the past for purposes such as analyzing the evolution of software systems [10]. In our case, we define a change cluster to be a set of program elements (methods or fields) that are related through their change history. Our general assumption is that a developer working on a task related to a change cluster can potentially benefit from knowledge about the set of elements in the cluster. Given this assumption, we were interested in estimating the potential value of change clusters for supporting program navigation. In other words, how often do tasks overlap with change clusters? To what degree does the retrieval of a change cluster produce valuable information for developers?

These questions build on previous research in repository mining for the purpose of software engineering. Others have proposed to mine software change repositories for association rules, and to *recommend* an element for investigation if it has consistently been changed together with an element currently being modified by the developer [17, 20]. Although this idea was shown to be very good at recommending specific elements in particular situations (i.e., systematic co-modifications of the same set of elements), it is too specialized to support general-purpose code investigation. Instead, our goal was to broaden the idea of mining association rules between sets of elements by proposing clusters of elements related through change history, but that were not necessarily modified in the same change sets.

To investigate the value of retrieving change clusters to assist program navigation, we implemented a fine-grained change clustering technique and applied it to the revision history of seven mature open-source systems. Our study of 3 500 change sets for these systems covered a cumulative

time span of close to 12 years of development. Our analysis of this data shows that less than 12% of the changes could have benefited from change clusters. However, our analysis also allowed us to make numerous insightful observations on the factors that influence how we can use change clusters to guide program navigation.

2. History-Based Clusters

The basic idea of task-related change clusters is to search the development history of a software project to detect groups of elements (clusters) that form a cohesive subset of the program. This approach broadens the idea of mining association rules between sets of elements [17, 20] by proposing clusters of elements related through change history, but whose change pattern is not strictly an instance of an association rule. The idea for searching for task-related clusters in revision history stems from our previous work on the reuse of program investigation knowledge for code understanding [13].

2.1. Background

Our proposed approach relies on the analysis of repositories storing the change history of software systems. Such repositories typically store software changes as differences between versions of artifacts. Central to our approach is the concept of a *transaction* (or change set), i.e., a number of software artifacts *committed* together to a software repository. Some repository software (such as Subversion) explicitly support the tracking of transactions. Other systems (such as CVS), do not. In the latter case, it is nevertheless possible to convert a stream of *commit* operations on individual artifacts into transactions. Following common practice for mining CVS repositories [19], we consider all commits sharing a user and log message performed during a given time window to constitute a transaction.

The *merging* of branches is another issue that arises when analyzing software repositories. This operation is not explicitly documented by neither CVS nor Subversion. Detecting merges is important because we do not want to analyze the same change twice: one in the branched version and one in the merged version. As in previous work [19], we avoid the issue of branch merging by filtering out large transactions (see Section 2.3).

Finally, although commit operations are performed at the granularity of files, a parsing operation will extract information about the individual program elements that were changed as part of a transaction. Henceforth, we will assume that a software repository can be abstracted as a sequence of transactions, each describing the set of *changed elements* (fields, methods). For each changed element, we record whether the element was *added*, *deleted*, or *modified* as part of the transaction.

2.2. Motivating Example

We illustrate the potential benefits of change clusters with a scenario taken from the change history of Apache Lucene,[1] a text search engine library.

In October 2003, a developer starts fixing a bug having to do with *locking* (Transaction #1105). The modification ends up requiring the developer to commit changes to seven methods in three distinct classes, and presumably to investigate a greater amount of code.

As it turns out, the code touched is related to a high-level "locking" concern, and code related to this concern was modified multiple times in the past. For instance, more than 18 months ago a different developer had committed a very similar set of elements with the log message "obtain write.lock while deleting documents" (Transaction #670). Very similar transactions were also committed months later by other developers (#1005, #1095). In fact, the union of all the 24 methods committed as part of transactions #670, 1005, and 1095 contains *all* the seven methods committed as part of #1105, and likely many methods also investigated (but not modified) by the developer. From this example, one can surmise that the ability to effortlessly retrieve the elements changed as part of the three above transactions would have been an asset to the developer. But how to find the set of related transactions? A textual search for the word "lock" in all the change logs yields over 30 distinct transactions, with a vast number of irrelevant elements. Association rule mining would also not have yielded this result, because the seven modified methods had never been modified together in the past.

The clustering technique we describe in the rest of this section is able to identify the three related transactions given *any combination of 2 methods in transaction #1105*. In our scenario, having identified two relevant methods, the developer would have been able to instantly view a *recommendation* consisting of the 22 methods in the matching change cluster.

2.3. Overview of the Technique

We model a program $P = \{e_1, ..., e_n\}$ as a set of program elements e_i, which in our case are Java fields and methods. Our technique takes as input a query $Q \subseteq P$ and returns a related cluster $C \subseteq P$.[2] The idea is that Q represents a small number of elements related to a task (typically 2 or 3), and that C represents a larger set of elements that are part of *clusterable transactions*, and that are related to the task.

[1] `lucene.apache.org/`

[2] In practice a cluster is a more elaborate data structure that retains the list of transactions composing it, but this level of detail is superfluous here.

Given a query, our technique retrieves relevant clusters in four steps:

1. Determine the sequence of clusterable transactions.
2. Cluster transactions.
3. Retrieve the clusters matching a query.
4. Filter retrieved clusters.

Determine the sequence of clusterable transactions. Transactions that involve too few or too many changed elements to be useful are removed from the list of transactions processed by the clustering algorithm (see next paragraph for thresholds). The result of this step is a list of *clusterable transactions*.

Cluster transactions. The clusterable transactions are clustered based on the number of overlapping elements using a standard nearest-neighbor clustering algorithm (Appendix A). We used a nearest-neighbor clustering algorithm because it is a simple and intuitive way to associate elements transitively through transactions. In other words, if elements A and B are changed together in one transaction, and then B and C are changed together in another transaction, A and C might have a relation worth reporting to a developer. During our initial investigation with this technique, we also did not encounter any issue that would justify or motivate experimentation with different clustering algorithms. The result of this clustering operation is a set of clusters representing transactions that address an overlapping set of elements. The only parameter of our clustering algorithm is the minimum number of common elements between two transactions required to assemble the two transactions in a cluster. Experimentation with the Lucene change history revealed 4 elements as the threshold leading to the most balanced results. To produce useful clusters, we remove, in the first step, all transactions with less than four elements from the sequence of clusterable transactions as they can never be clustered. Additionally, based on prior experimentation [13], we also remove transactions with more than 20 changed elements as these generate overly large clusters that would require developers to spend an unreasonable amount of effort to study. Removing large transactions has the added side-effect of eliminating transactions representing branch merges.

Retrieve the clusters matching a query. We return all the clusters $C \in \mathcal{C} | Q \subseteq C$.

Filter retrieved clusters. Based on various filtering heuristics (Section 2.4), we remove the clusters that are not likely to be useful to developers from the list of results.

2.4. Filtering Heuristics

The results of the search technique described above can be influenced by a number of heuristics that are applied to the four steps above. We experimented with a number of filtering heuristics that we developed based on insights gained during a preliminary study [13], and through extensive iterative experimentation with the change history of the Lucene system described in Section 2.2. Based on these insights, we fixed a number of parameters that clearly appeared as offering good results, such as the minimum overlap value of 4 for the clustering algorithm. Other heuristics required further experimentations and remained variables in our study.

Heuristic 1 (Ignore High Frequency) *In step 3, queries automatically return no result if any of the query elements is an element that changed more than a specified number of times as part of a transaction.*

We define the function

$$\text{highFrequency}(e, T) \rightarrow \text{boolean}$$

as returning **true** if element e occurs in 3% or more of the transactions in T. We designed this heuristic through prior experimentation after noticing many situations where some central or critical elements were continually modified [13]. In such cases, our hypothesis is that querying the change history for the highly changed element returns too many recommendations to be useful.

Heuristic 2 (Minimum Cohesion) *We define the cohesion of a cluster C created through the clustering of n transactions T_i as $\frac{\sum_{i=1}^{n} |T_i|}{|C| n}$. Cohesion varies between 0 and 1 and measures how much the transactions clustered actually overlap. For example, a cluster created from transactions grouping identical sets of changed elements would have a cohesion of 1.0. Two transactions of 5 elements, of which 4 overlap, would create a cluster of cohesion $(5 + 5)/(6 \cdot 2) = 0.83$.*

The intuition behind this heuristic is that clusters with low cohesion may represent transactions that have been clustered but that do not represent changes associated with a common high-level concern. Based on prior experimentation and on an analytic interpretation of the measure, we determined that 0.6 seemed a reasonable cohesion cutoff. When this heuristic is enabled, in step 3 of the technique, clusters are only returned if they have a minimum cohesion of 0.6.

Heuristic 3 (Minimum Transactions) *In step 3, the minimum number of transactions that must be associated with a cluster for it to be returned as a match.*

The idea behind this last heuristic is to avoid returning results that are single transactions or very small groups of transactions, which may have been spuriously clustered. We experimented with values 2 and 3. A value of one returns all clusters, whereas higher values produce very few recommendations.

3. Empirical Assessment

The overall goal of this research was to assess *to what extent can we use past changes to inform program navigation?* This section describes the empirical setup we designed to answer that question.

3.1. Research Question

We refine our general research goal with three specific research questions:

Q1. Support Frequency. What percentage of change tasks relate to history-based clusters? Although we expect support frequency to vary across different projects, we wanted to get a general estimate of the amount of historical information we can to use to produce recommendations to support ongoing maintenance.

Q2. Impact of Heuristics. What is the impact of the filtering heuristics on the usefulness of the technique?

Q3. Value of the information. To what degree are recommended clusters likely to be useful to developers?

3.2. Methodology

The basic methodology we employed for answering the research questions was to apply various configurations of our technique to the change repositories of a number of long-lived Java systems. For each system we proceeded with the following steps:

1. We produced a sequence of *analyzable transactions*. An analyzable transaction is a transaction with three or more changes that are *not* additions (i.e., that are changed or deleted elements), and with a total of 20 or fewer elements. Transactions outside this range cannot be efficiently analyzed with the procedure described in the following steps (because newly added elements have no chance of resulting in a matching query unless they are reintroduced). The list of analyzable transactions is different from the list of clusterable transactions in that analyzable transactions represent transactions to which our empirical design is applicable, whereas clusterable transactions are any transactions that can be used to create clusters.

2. We skip the first 200 analyzable transactions, to analyze transactions that approximate tasks on a system with a reasonable amount of change history. We analyze the following 500 analyzable transactions in the system's change history.

3. For each *analyzable* transaction T_i (for $i > 200$), we apply the clustering algorithm of Appendix A to the set of all *clusterable* transactions $\{T_j \mid j < i\}$.

4. For transaction T_i, we produce a set of queries $\mathcal{Q} = \{(e_m, e_n) | (e_m, e_n) \in T_i \times T_i \wedge e_m \neq e_n \wedge \{e_m, e_n\} \cap \text{additions}(T_i) = \emptyset\}$. In other words, we consider all possible combinations of two elements in T_i that do not correspond to additions. Although, in theory, $|\mathcal{Q}| = \binom{|T_i|}{2}$ can grow very big, restricting our analysis to transactions with a maximum of 20 (changed) elements leads to a tractable number of combinations (190). We use this strategy because a maintenance task can be approached from different angles (i.e., different elements being selected as starting points for the investigation). For example, a developer might want to start with elements A and B while another developer would start by looking at elements B and C. With our strategy, we exhaustively take into account all possible starting points of two elements.

5. We retrieve the clusters formed by at least 2 transactions. If filtering heuristics are enabled, we apply the heuristics to remove unwanted clusters.

6. We measure various properties of the output clusters. We refer to the output of our technique as *recommended clusters*, or simply *recommendations*.

We use four measures to assess the results of experiments as described above.

Measure 1 (Supported Transactions) *The number of transactions (out of 500 analyzed for a system) for which at least one query generates a recommendation.*

This measure assesses how many tasks could *potentially* have benefited from the technique, in answer to research question Q1.

Measure 2 (Input Coverage) *For transactions producing at least one recommendation, the ratio of recommendations to the total number of queries for the transaction. For instance, a transaction with 5 elements would generate* $\binom{5}{2} = 10$ *queries. If only 4 of these queries produce a recommendation, Input Coverage* $= 0.4$*. This value is aggregated over all supported transactions to produce an overall ratio.*

This measure estimates the probability that a transaction would have benefited from the technique if a developer entered a query based on the first two relevant elements identified. In other words, if for a transaction *Input Coverage* $= 1.0$, any query will produce the output cluster, and hence the probability to produce a recommendation is 1. The measure of input coverage will help us assess Q1.

Measure 3 (Scattered Clusters) *The ratio of transactions for which there is at least one recommendation of a scattered cluster to the total number of supported transactions. A scattered cluster is defined as a cluster grouping elements in at least 3 different classes and 2 different packages, generated from transactions that span at least seven days.*

This measure estimates the number of tasks for which recommended clusters could have been particularly useful to developers as they represent scattered (and thus potentially hard to find) elements. This measure is intended to help provide answers to questions Q2 and Q3.

Measure 4 (Recommendation Accuracy) *A measure of how accurate the recommended cluster is. We estimate the accuracy of a recommended cluster by calculating the average precision and recall of each recommendation, and then generating the average F-measure for all the recommendations. The precision for a cluster is the number of non-query elements in the cluster that are also in the transaction analyzed, divided by the number of non-query elements in the cluster. The recall is the number of non-query elements in the cluster that are also in the transaction, divided by the number of non-query elements in the transaction. The F-measure is the harmonic mean of both precision and recall, calculated as* $F = 2 \cdot P \cdot R/(P + R)$. *Our final measure of recommendation accuracy is the average F measure over all recommendations. In brief, Recommendation Accuracy varies between 0 and 1: a value of 1 indicates that precision and recall are perfect for all recommendations, and degrades according to a corresponding degradation in precision/recall as measured by the F-measure.*

In the context of our approach, it should be noted that the precision and recall ratios grossly *under*estimate the performance of the approach because they are calculated based on the elements *changed* as part of a task, whereas we attempt to produce recommendations of what a developer needs to *investigate*. We have used the accuracy measure described above because, in the absence of additional information about the change tasks, it is the only way to rigorously assess our technique. In practice the F-measure calculated represents an extreme lower bound on the accuracy of the approach. For example, a recommendation with an F-measure of 0.0 may still have been useful if it allowed the developer to find related, but unchanged, program elements. The main purpose of this measure is to provide a robust and objective way to measure the impact of the various heuristics to answer Q2. However, the intermediate F-measures can also help us interpret the results for the purpose of assessing Q3.

3.3. Example

We clarify our experimental procedure and measures with an example. Figure 1 represents a hypothetical transaction stream and some clusters, and where each changed element is represented by a letter. We assume that all the changed elements represent modifications to existing program elements (as opposed to addition of new elements or deletion of existing elements).

According to our definitions all transactions are both analyzable and clusterable. We proceed with our methodology, using a clustering overlap of 2 to shorten the example (we use 4 in practice).

$T_1 = \texttt{A B C D}$ $\quad C_1 = \texttt{A B C D}$
$T_2 = \texttt{A B E F}$ $\quad C_2 = \texttt{A B C D E F}$
$T_3 = \texttt{G H I J}$ $\quad C_3 = \texttt{G H I J}$
$T_4 = \texttt{A B C I}$

Figure 1. Transaction stream and clusters

T_1**:** Analyzing T_1 produces no recommendation because it is the first transaction.

T_2**:** Analyzing T_2 generates a single cluster $C_1 = T_1$. For T_2,$\mathcal{Q} = \{(A, B), (A, E), (A, F), (B, E), (B, F), (E, F)\}$. Only (A, B) retrieves cluster C_1, which is filtered out because it only "clusters" a single transaction and MIN TRANSACTIONS = 2. We thus have no recommendation for T_2 as well.

T_3**:** Analyzing T_3 generates cluster C_2. Of the 6 possible queries for T_3, there is no overlap with the cluster.

T_3**:** Analyzing T_4 generates clusters C_2 and C_3. Queries (A, B), (A, C), and (B, C) all retrieve cluster C_2, which is considered a result because it represents 2 transactions. We thus consider T_4 to be a *supported transaction*. The *Input Coverage* metric for T_4 is thus $3/\binom{4}{2} = 0.5$. The precision of query (A, B) is $|\{C\}|/|\{C, D, E, F\}| = 0.25$, and the recall is $|\{C\}|/|\{C, D\}| = 0.5$, yielding $F = 0.33$. The results are the same for the other two queries, yielding a final *Recommendation Accuracy* of 0.33.

4. Results

We describe our data sources and analyze the results of the experiments to answer our three research questions.

4.1. Data Sources

Systems were selected among the population of available and long-lived open-source Java projects. To be selected for analysis, a system needed to have at least 700 analyzable transactions. For each system, we considered all transactions starting at the first transaction, but removed small and large transactions as described in the previous section. Table 1 reports on the systems we analyzed and their main characteristics as related to our research questions. References to the systems analyzed are in Appendix B. For each system, the columns *First* and *Last* give the dates of the first and last analyzed transactions, respectively. These correspond to the 201st and 700th analyzable transactions available in the repository. The following column gives the number of days between the first and last transactions analyzed. Column *Trans.* gives the total number of transactions committed between the first and last analyzed transactions.

System	First	Last	Time	Trans.	Clusters	Pool	Ratio
Ant	30 Oct 2001	10 Dec 2004	1136	2353	860	962	1.12
Azureus	11 Nov 2003	17 May 2004	188	2377	711	814	1.14
Hibernate	21 Nov 2003	9 Jun 2005	565	2237	868	952	1.10
JDT-Core	24 Jan 2002	21 Feb 2003	393	3216	557	684	1.23
JDT-UI	16 Aug 2001	18 Apr 2002	245	2555	765	823	1.08
Spring	13 Jan 2004	30 Nov 2004	322	1578	785	912	1.16
Xerces	7 May 2001	2 May 2005	1455	2189	665	769	1.16
Total			4304	16 505			

Table 1. Characteristics of Target Systems

System	Supported	Ratio	Coverage
Ant	87	0.17	0.18
Azureus	113	0.23	0.26
Hibernate	69	0.14	0.23
JDT-Core	163	0.33	0.25
JDT-UI	47	0.09	0.16
Spring	109	0.22	0.28
Xerces	89	0.18	0.37
Average	96.7	0.19	0.25

Table 2. Support and Input Coverage

For instance, for the Ant project, we analyzed 500 transactions out of 2353 because $2353 - 500 = 1853$ transactions were to small or too large to be considered analyzable (see Section 3.2). Finally, the last three columns describe the amount of clustering that took place for the last analyzed transaction. Column *Clusters* gives the clusters formed for the last analyzable transaction and column *Pool* gives the maximum number of clusterable transactions. The last column is the ratio of Pool to Clusters, or the average number of transactions per cluster. Experimentation showed that this ratio tends to remain stable (i.e., the number of clusters grows linearly with the number of transactions analyzed).

4.2. Q1: Support Frequency

Table 2 reports on the measure of *Supported Transactions* and *Input Coverage* for all systems analyzed. The second and third columns give the number of supported transactions (out of 500) and the corresponding ratio, respectively. The last column gives the input coverage measure.

As this data shows, on average 1 in 5 analyzed transaction overlaps with change clusters. As it is expected, the number of supported transactions varies per system (σ=0.076 for the ratio), with a distribution between the extremes that is close to uniform. The input coverage values show that on average 1 in 4 queries will yield a recommended cluster if a cluster can be recommended. Values of input coverage metric are more stable (σ=0.069).

The general answer to our first research question is that it is reasonable to expect that around 1 in 5 maintenance tasks are associated with change clusters. In addition, querying for change clusters based on two relevant methods will identify the cluster 1 in 4 times. A rough overall interpretation of this data is thus that a developer who identifies two elements modified as part of a task and performs a query using our technique can expect to see a recommendation $0.19 \times 0.25 \approx 5\%$ of the time.

4.3. Q2: Effect of Filtering Heuristics

To study the effect of the various heuristics, we calculated the effect of applying the heuristic on the values of the *Supported Transactions*, *Scattered Clusters*, and *Accuracy* metrics. We can analytically determine that *Supported Transactions* will drop as the result of filtering more clusters: our experiments revealed to what extent. In the case of *Scattered Clusters*, we have no a priori theory about the effect of the heuristics on the nature of the clusters identified: we used our experiments to discover this effect (using a two-tailed Wilcoxon signed-rank test).[3] Finally, the heuristics were designed to improve the results, so our theory was that the heuristics should result in an increased accuracy, and we used our experiments to test this hypothesis using 1-tailed Wilcoxon signed-rank test. In the following, we consider results to be statistically significant at p=0.05.

Table 3 reports on the measure of *Supported Transactions*, *Scattered Clusters*, and *Accuracy* for the default configuration. In this configuration, Heuristics 1 (IGNORE HIGH FREQUENCY) and 2 (MIN COHESION 0.6) are disabled, and the parameter value for Heuristic 3 (MIN TRANSACTIONS) is 2. No correlation was detected between the *Supported Transactions*, *Scattered Clusters*, and *Accuracy* variables.

Table 4 shows the effect of the IGNORE HIGH FREQUENCY heuristic. Applying this heuristic results in an average 7.8% drop in the number of supported transactions. There is no statistically significant effect on the number of scattered clusters reported. A one-tailed Wilcoxon test shows a statistically significant (positive) impact on the accuracy (p=0.0178).

[3]The Wilcoxon signed-rank test is more conservative than the *t*-test. It does not assume normality of difference scores.

System	Supported	Scattered	Accuracy
Ant	87	0.37	0.28
Azureus	113	0.47	0.28
Hibernate	69	0.33	0.29
JDT-Core	163	0.64	0.34
JDT-UI	47	0.55	0.31
Spring	109	0.51	0.36
Xerces	89	0.56	0.42
Average	96.7	0.49	0.33

Table 3. Measures – Default Configuration

The impact of IGNORE HIGH FREQUENCY on the *scattered clusters* metric for Hibernate is an outlier (-42%). We manually inspected the recommended clusters to find an explanation for this value. As it turns out, 100% of the 12 transactions filtered by the heuristic had recommended a scattered cluster. However, in all cases the high-frequency element causing the rejection was either a factory method or the root of a data structure. Because of the "global" nature of these elements in the design of Hibernate, their modification together with other elements inevitably classified a cluster as "scattered". Despite this aberration, the heuristic increased the overall accuracy of the results. We therefore consider IGNORE HIGH FREQUENCY to be a generally valuable heuristic.

System	Supported	Scattered	Accuracy
Ant	-5%	+5%	+3%
Azureus	-14%	+1%	+4%
Hibernate	-17%	-42%	+8%
JDT-Core	-4%	-1%	+3%
JDT-UI	-6%	+7%	+2%
Spring	-2%	+2%	0%
Xerces	-7%	-8%	+6%
Average	-7.8%	-5.2%	+3.6%

Table 4. Effect of Ignore High Frequency

Table 5 shows the effect of the MINIMUM COHESION (0.6) heuristic. Applying this heuristic results in an overall 45.7% drop in the number of supported transactions. A two-tailed Wilcoxon test shows a statistically significant negative effect on the number of scattered clusters reported (p=0.0156). There is no statistically significant positive impact on the accuracy. The negative effect on the number of scattered clusters report is easily explained by the design of this heuristic. If we assume that most transactions tend to modify elements in the same classes, then requiring a minimum of cohesion will eliminate overly scattered clusters. This heuristic does not provide a clear advantage: although it might increase the accuracy, it may result in simply recommending more obvious information.

System	Supported	Scattered	Accuracy
Ant	-46%	-42%	+13%
Azureus	-11%	-18%	+33%
Hibernate	-68%	-32%	+12%
JDT-Core	-32%	-41%	+6%
JDT-UI	-64%	-26%	-22%
Spring	-46%	-11%	+28%
Xerces	-54%	-48%	+29%
Average	-45.7%	-31.0%	+14.3%

Table 5. Effect of Minimum Cohesion 0.6

Table 6 shows the effect of the MINIMUM TRANSACTIONS (3) heuristic. Applying this heuristic results in an average 56% drop in the number of supported transactions. A two-tailed Wilcoxon test shows a statistically significant positive effect on the number of scattered clusters reported (p=0.0156). There is no statistically significant positive impact on the accuracy. Again, the impact on *Scattered Clusters* is intuitive: clusters with more transactions are likely to be more scattered. The fact that this heuristic has no obvious bearing on accuracy also means that it might not be universally desirable.

System	Supported	Scattered	Accuracy
Ant	-71%	+31%	+8%
Azureus	-53%	+57%	-4%
Hibernate	-42%	+5%	+3%
JDT-Core	-45%	+26%	+1%
JDT-UI	-77%	+15%	+55%
Spring	-64%	+25%	-28%
Xerces	-40%	+21%	-20%
Average	-56%	+26%	+2%

Table 6. Effect of Min Transactions 3

The conclusion of our experiment is that the IGNORE HIGH FREQUENCY heuristic should always be applied, and that the two other heuristics should be chosen based on the characteristics of the desired results. MIN COHESION 0.6 helps produce more accurate but less scattered results. MIN TRANSACTIONS 3 is more aggressive, producing results that are potentially more useful, but also may be irrelevant more often.

4.4. Q3: Result Usefulness

Because our overall accuracy metric is difficult to interpret, we provide a simpler metric: the ratio of supported transactions for which the recommended cluster overlapped with the task (with at least one element in addition to the query). Such transactions correspond to supported transaction with an average F-measure greater than zero. Table 7 presents the results for the default configuration (Def.) as

System	Def.	HF	MC-0.6	MT-3
Ant	48%	50%	47%	52%
Azureus	54%	59%	19%	53%
Hibernate	58%	63%	73%	48%
JDT-Core	56%	58%	54%	53%
JDT-UI	49%	52%	59%	36%
Spring	61%	62%	61%	69%
Xerces	58%	61%	56%	53%
Average	55%	58%	53%	52%

Table 7. Overlap

well as for IGNORE HIGH FREQUENCY (HF), MIN COHESION 0.6 (MC-0.6), and MIN TRANSACTIONS 3 (MT-3). As expected from the results in Section 4.3 the only heuristic with a consistent positive impact is IGNORE HIGH FREQUENCY.

We observe that, on average, the recommended cluster for 58% of the supported transactions overlaps with the elements modified as part of the task. Conservatively, we can interpret this value to be an upper bound on the number of tasks that have a measurable potential of being supported through our change clusters. Combining this ratio with the average support ratio in Table 2, we determine that we cannot reasonably expect that more than $0.19 \times 58\% = 11\%$ of the tasks have the *potential* to benefit from change clusters. We conclude that, although cases such as the one described in Section 2.2 have good potential to decrease the amount of program navigation required for a task, such cases will not be common in the life-cycle of a system. Future developments with our or similar approaches should thus focus on maximizing the quality of the match between the current task and past transactions, rather than finding many potential matches.

Although a systematic qualitative study of the 677 supported transactions we detected is beyond the scope of this paper, we conducted a partial study of the supported transactions. For each system, we inspected the supported transactions with the highest F-measure, and two randomly-selected transactions with an F-measure of 0. Although it does not provide definite answers, this initial inspection allowed us to make a number of useful observations that will help guide future research and tool development.

- Initial evidence indicates a good correlation between our measure of scattering for a cluster and the actual usefulness of the cluster. Clusters generating very high F-measures typically group transactions that are very close in time, not only to each other but also to the analyzed task. The benefit of recommending such clusters is limited, and our definition of scattering appears to be a promising filtering metric.
- Some recommendations are clearly spurious and the result of continuous modifications to unstable code. Transactions addressing code cleanups, simple refactorings, javadocs, etc., are unlikely to generate useful recommendations independently of the retrieval technique used.
- Valuable clusters include instances of association rules, or instances of a rule with small exceptions. For example, in Azureus, there is a near-perfect association rule related to the concern "connection closing". Two developers were involved and the modified lines were almost the same across the transactions of the recommended clusters. Such evidence seems to indicate that it is sensical to provide recommendations based on imperfect matches to association rules.
- Many of the zero-overlap recommendations we inspected were caused by central classes that changed often, hence providing qualitative evidence that the IGNORE HIGH FREQUENCY heuristic is useful.
- In cases when there was a very large time span between transactions in a cluster, the earliest transactions were not relevant because the concern modified by the latest transaction did not exist. A more sophisticated implementation of this technique should probably include a component to remove obsolete transactions.

4.5. Threats to Validity

In designing our study we favored a quantitative focus that would allow us to analyze massive amounts of historical data over a design favoring a more in-depth analysis of the recommendations. As a result, our measure of recommendation accuracy is highly abstracted, and may not always accurately reflect the actual usefulness of recommended clusters.

In our overall evaluation of recommendation accuracy, we have also made three conservative assumptions. First, we have systematically queried for recommendations for *all* analyzable tasks, including tasks for which a developer would not require assistance (e.g., introducing localization strings, adding comments, etc.). Second, we compared our recommendations against elements actually modified as part of a task, even though we know that these form a proper subset of the elements investigated as part of the task. Finally, since we have not incrementally re-created intermediate versions of the system, we could not detect the recommended elements that no longer existed at the time of the corresponding query. In practice, such elements would be eliminated from the recommendations, hence improving the precision of the technique.

5. Related Work

There exists extensive previous work on both the mining of software repositories and on the use of clustering algorithms in software engineering. This discussion focuses on the most similar and recent work in the area of software evolution.

Mining Software Repositories

Our technique was partially inspired by the work of Zimmermann et al. [20] and Ying et al. [17] on the mining of association rules in change history. As described in Section 1, we sought to expand the technique to be able to recommend larger (but less precise) clusters of elements to guide program navigation.

Bouktif et al. also investigated how to recommend co-changes in software development [2]. As opposed to the work cited above, Bouktif et al. used change patterns instead of association rules. Also, their approach does not attempt to reconstruct transactions, and can consider associated files that were changed in different transactions.

ChangeDistiller is a tool to classify changes in a transaction into fine-grained operations (e.g., addition of a method declaration), and determines how strongly the change impacts other source code entities [6]. Our approach uses similar repository analysis techniques but is focused on providing task-related information as opposed to an overall assessment of a system's evolution.

Finally, repository mining can also be used to detect *aspects* in the code [3]. In this context, aspects are recurring sets of changed elements that exhibit a regular structure. Aspects differ from the clusters we detect in the regular structure they exhibit, which may not necessarily align with the code that is investigated as part of change tasks.

Clustering Analysis

The classical application of clustering for reverse engineering involves grouping software entities based on an analysis of various relations between pairs of entities of a given version of the system [8]. Despite its long and rich history, experimentation with this approach continues to this day. For example, Andreopoulos et al. combined static and dynamic information [1], Kuhn et al. used a textual similarity measure as the clustering relation [11], and Christl et al. used clustering to assist iterative, semi-automated reverse engineering [4]. The main differences between most clustering-based reverse engineering techniques and the subject of our investigation is that the entities we cluster are transactions, rather than software entities in a single version of a system. For this reason, our analysis is based strictly on the evolving parts of the system.

Both Kothari et al. [10] and Vanya et al. [15] recently reported on their use of clustering to study the evolution of software systems. The idea of using change clusters is the same in both works and ours, but the purpose of the work is different. Kothari et al. use change clusters to uncover the types of changes that happened (e.g., feature addition, maintenance, etc.) during the history of a software system. Vanya et al. use change clusters (which they call "evolutionary clusters") to guide the partitioning of a system that would increase the likelihood that the parts of the system would evolve independently. In contrast, we cluster transactions based on overlapping elements (not files), to recommend clusters to support program navigation, as opposed to architectural-level assessment of the system.

Finally, Hassan and Holt evaluated, on five open source systems, the performance of several methods to indicate elements that should be modified together [7]. This study found that using historical co-change information, as opposed to using simple static analysis or code layout, offered the best results in terms of recall and precision. The authors then tried to improve the results using filtering heuristics and found that keeping only the most frequently co-changed entities yielded the best results. As opposed to our approach, the evaluated filtering heuristics were only applied on entities recovered using association rules and not using clustering techniques. The focus of their study was also more specific, as they recommend program elements that were strictly *changed*, as opposed to recommending elements that might be *inspected* by developers.

6. Conclusion

Developers often need to discover code that has been navigated in the past. We investigated to what extent we can benefit from change clusters to guide program navigation. We defined change clusters as groups of elements that were part of transactions (or change sets) that had elements in common. Our analysis of close to 12 years of software change data for a total of seven different open-source systems revealed that less than 12% of the changes we studied could have benefited from change clusters. We conclude that further efforts should thus focus on maximizing the quality of the match between the current task and past transactions, rather than finding many potential matches. Our study has already helped us in this goal by providing reliable evidence of the effectiveness of some filtering heuristics, and useful insights for the development of additional heuristics.

Acknowledgments

The authors thank Emily Hill and José Correa for their advice on the statistical tests, and the anonymous reviewers for their helpful suggestions. This work was supported by NSERC.

References

[1] B. Andreopoulos, A. An, V. Tzerpos, and X. Wang. Multiple layer clustering of large software systems. In *Proc. 12th Working Conf. on Reverse Engineering*, pages 79–88, 2005.

[2] S. Bouktif, Y.-G. Guéhéneuc, and G. Antoniol. Extracting change-patterns from cvs repositories. In *Proc. 13th Working Conf. on Reverse Engineering*, pages 221–230, 2006.

[3] S. Breu and T. Zimmermann. Mining aspects from version history. In *Proc. 21st IEEE/ACM Int'l Conf. on Automated Software Engineering*, pages 221–230, 2006.

[4] A. Christl, R. Koschke, and M.-A. Storey. Equipping the reflexion method with automated clustering. In *Proc. 12th Working Conf. on Reverse Engineering*, pages 89–98, 2005.

[5] D. Čubranić, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, 2005.

[6] B. Fluri and H. C. Gall. Classifying change types for qualifying change couplings. In *Proc. 14th IEEE Int'l Conf. on Program Comprehension*, pages 35–45, 2006.

[7] A. E. Hassan and R. C. Holt. Replaying development history to assess the effectiveness of change propagation tools. *Empirical Software Engineering*, 11(3):335–367, 2006.

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

[9] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *Proc. 2nd Int'l Conf. on Aspect-Oriented Software Development*, pages 178–187, 2003.

[10] J. Kothari, T. Denton, A. Shokoufandeh, S. Mancoridis, and A. E. Hassan. Studying the evolution of software systems using change clusters. In *Proc. 14th IEEE Int'l Conf. on Program Comprehension*, pages 46–55, 2006.

[11] A. Kuhn, S. Ducasse, and T. Gîrba. Enriching reverse engineering with semantic clustering. In *Proc. 12th Working Conf. on Reverse Engineering*, pages 133–142, 2005.

[12] M. P. Robillard. Topology analysis of software dependencies. *ACM Transactions on Software Engineering and Methodology*, 2008. To appear.

[13] M. P. Robillard and P. Manggala. Reusing program investigation knowledge for code understanding. In *Proc. 16th IEEE Int'l Conf. on Program Comprehension*, pages 202–211, 2008.

[14] J. Sillito, G. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *Proc. 14th ACM SIGSOFT Int'l Symposium on the Foundations of Software Engineering*, pages 23–34, 2006.

[15] A. Vanya, L. Hofland, S. Klusener, P. van de Laar, and H. van Vliet. Assessing software archives with evolutionary clusters. In *Proc. 16th IEEE Int'l Conf. on Program Comprehension*, pages 192–201, 2008.

[16] N. Wilde and M. C. Scully. Software reconnaissance: Mapping program features to code. *Software Maintenance: Research and Practice*, 7:49–62, 1995.

[17] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

[18] A. Zeller. The future of programming environments: Integration, synergy, and assistance. In *Proceedings of the 29th International Conference on Software Engineering—The Future of Software Engineering*, pages 316–325, 2007.

[19] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proc. 1st Int'l Workshop on Mining Software Repositories*, pages 2–6, May 2004.

[20] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proc. 26th ACM/IEEE Int'l Conf. on Software Engineering*, pages 563–572, 2004.

A. Clustering Algorithm

This algorithm is not sensitive to whether a given program element exists or not in a given version of a program. For example, if method m exists in one version, it is considered a valid program element even if it is removed in a later version. In the rest of this section, we use the term "program element" to refer to the uniquely identifying representation of the element (e.g., a Java fully-qualified name).

Let T be a transaction modeled as a set of program elements changed together during the history of a software system. Let $\mathcal{T}$ be a sequence of transactions. In this algorithm a cluster is also modeled as a set of elements.

1: **Input**: $\mathcal{T}$: A sequence of transactions
2: **Parameter**: MINOVERLAP: A positive, non-zero value indicating the minimum overlap between two transactions in a cluster
3: **Var**: $\mathcal{C}$: A set of clusters, initially empty.
4: **for all** $T_i \in \mathcal{T}$ **do**
5: MaxOverlap $\leftarrow 0$
6: MaxIndex $\leftarrow -1$
7: **for all** $C_j \in \mathcal{C}$ **do**
8: **if** $|C_j \cap T_i| >$ MaxOverlap **then**
9: MaxOverlap $\leftarrow |C_j \cap T_i|$
10: MaxIndex $\leftarrow j$
11: **end if**
12: **end for**
13: **if** $(\text{MaxIndex} >= 0) \wedge$ $(\text{MaxOverlap} \geq \text{MINOVERLAP})$ **then**
14: $C_{\text{MaxIndex}} \leftarrow (C_{\text{MaxIndex}} \cup T_i)$
15: **else**
16: NewCluster $\leftarrow T_i$
17: $\mathcal{C} \leftarrow \mathcal{C} \cup \{$ NewCluster $\}$
18: **end if**
19: **end for**
20: **return** $\mathcal{C}$

B. Systems Analyzed

System home pages last verified 7 May 2008.

Ant	`ant.apache.org/`
Azureus	`azureus.sourceforge.net/`
Hibernate	`www.hibernate.org/`
JDT-Core	`www.eclipse.org/jdt/core/`
JDT-UI	`www.eclipse.org/jdt/ui/`
Spring	`springframework.org/`
Xerces	`xerces.apache.org`