JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2010; **22**:143–164
Published online 9 September 2009 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.413

Research

Recommending change clusters to support software investigation: an empirical study

Martin P. Robillard*, † and Barthélémy Dagenais

School of Computer Science, McGill University, Montréal, QC, Canada

SUMMARY

During software maintenance tasks, developers often spend a valuable amount of effort investigating source code. This effort can be reduced if tools are available to help developers navigate the source code effectively. We studied to what extent developers can benefit from information contained in clusters of change sets to guide their investigation of a software system. We defined change clusters as groups of change sets that have a certain amount of elements in common. Our analysis of 4200 change sets for seven different systems and covering a cumulative time span of over 17 years of development showed that less than one in five tasks overlapped with change clusters. Furthermore, a detailed qualitative analysis of the results revealed that only 13% of the clusters associated with applicable change tasks were likely to be useful. We conclude that change clusters can only support a minority of change tasks, and should only be recommended if it is possible to do so at minimal cost to the developers. Copyright © 2009 John Wiley & Sons, Ltd.

Received 29 January 2009; Revised 12 June 2009; Accepted 16 July 2009

KEY WORDS: software clustering; mining software repositories; recommendation systems

1. INTRODUCTION

When involved in a task to change unfamiliar code, a software developer will generally spend a valuable fraction of the task time investigating the code. In many development environments,

*Correspondence to: Martin P. Robillard, School of Computer Science, McGill University, Montréal, QC, Canada.
†E-mail: martin@cs.mcgill.ca

Contract/grant sponsor: Natural Sciences and Engineering Council of Canada (NSERC)

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 [1]. Examples of advanced tools and techniques to support software investigation include query-based source code browsers [2], association rule mining of change history [3], searchable project memory [4], automated feature location [5], topology analysis of software dependencies [6], and semantic analysis [7]. The rich and diverse collection of available software investigation tools and techniques is not surprising when we consider the wide variety of questions developers ask during software change tasks [8]. 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.

To complement this corpus, we were interested in assessing the potential of a software's change history to guide developers in their exploration of the code in the context of a change task. Specifically, we sought to determine if we could *recommend* useful information about past changes to a developer to facilitate their exploration of the code. For this purpose, we devised a simple technique that scans the change history of a software system and determines if there exists any *change clusters* that overlap with a set of elements of interest to a developer. The concept of a change cluster has been used in the past for purposes such as analyzing the evolution of software systems [9]. In our case, we define a change cluster to be a set of fine-grained 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. However, to assess this potential, we must answer two important questions. First, how often do tasks overlap with change clusters? Second, to what degree does the retrieval of a change cluster produce valuable information for developers? In our investigation of these questions, we modeled change tasks as the set of elements committed together to a revision control system (see Section 2.1).

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 [3,10]. Although this idea was shown to be good at recommending specific elements at the granularity of classes in particular situations (i.e., systematic co-modifications of the same set of elements), it is too specialized to support general-purpose code investigation because there may exist cases where elements that should be investigated together were not changed in precisely the same change set. Instead, our goal was to broaden the idea of mining association rules between sets of elements by proposing clusters of fine-grained elements related through change history, but that were not necessarily modified in the same change sets.

We implemented a fine-grained change clustering technique and applied it to the revision history of seven mature open-source systems. Our study of 4200 change sets for these systems covered a cumulative time span of over 17 years of development. We analyzed this data in two ways. First, we performed a quantitative analysis to determine the degree of overlap between change tasks and change clusters, and tested different heuristics designed to increase the precision of the results (i.e., the relative number of recommended clusters that overlapped with a task). In a second phase of our investigation, we conducted an in-depth qualitative analysis of the recommended change clusters to assess their usefulness.

DOI: 10.1002/smr

2. CHANGE CLUSTERS

In our investigation, change clusters correspond to sets of fine-grained program elements (fields or methods) that were changed or added as part of overlapping change sets, and that form a cohesive subset of the program. The goal of extracting change clusters broadens the idea of mining association rules between sets of elements [3,10] by proposing clusters of elements also related through change history, but whose change pattern is not strictly an instance of an association rule. The motivation for searching for task-related clusters in the change history of a system stems from our previous work on the reuse of program investigation knowledge for understanding code [11].

2.1. Background

Extracting change clusters requires access to software repositories that store software changes as differences between versions of artifacts. Central to our study is the concept of a *transaction* (or change set), that is, a number of software artifacts *committed* together to a software repository. Some repository software (such as Subversion) explicitly supports 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 the common practice for mining CVS repositories [12], we consider all commits sharing a user and log message performed during a given time window to constitute a transaction.

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 and methods). For each changed element, we record whether the element was *added*, *deleted*, or *modified* as part of the transaction. In addition, transactions typically store meta-data such as a timestamp, the user name of the developer performing the transaction, and a log message.

In modeling change tasks, we chose to use transactions instead of all the changed elements associated with a feature or bug, as could be obtained from a bug database. Two main reasons motivated this decision. First, for some projects it can be impossible to track all the changes associated with a bug report because the tagging information is simply missing. Second, although we realize that there is generally a one-to-many relationship between tasks as defined by a bug report and transactions as recorded in a software repository, we were interested in studying interactions between fine-grained bursts of developer activity, which we believe are better represented by transactions than by the entire set of changes associated with a feature. This view is consistent with some of the prior work on mining software repositories [3].

2.2. Motivating example

We illustrate the potential benefits of change clusters with a scenario taken from the change history of Eclipse's Core plug-in (JDT-Core, see Table I). In JDT-Core, transaction #2416 is a bug fix about a property that stores the position in a source file of an abstract syntax tree node (the transaction

DOI: 10.1002/smr

Table I. References for target systems References (verified 6 January 2009).

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

numbers are internally generated by our analysis infrastructure). This property is set by calling the method `setSourceRange()` and the fix for the bug mainly involved modifying the method `setTypeForVariableDeclarationExpression()` in the class `ASTConverter`.

As it turns out, two previous transactions in JDT-Core's change history are highly related to this bug fix: #1470 and #1476. In transaction #1470, a developer added a call (and the supporting code) to `setSourceRange()` for array nodes in `ASTConverter.setTypeForVariableDeclarationExpression()` and introduced a similar code pattern in the method `ArrayType.clone()`. In transaction #1476, a developer removed the code in `ArrayType.clone()` and inlined it in `ASTConverter.setTypeForVariableDeclarationExpression()`. The developer also created a class, `CopyPositionsMatcher`, and used it in `ASTConverter.setTypeForVariableDeclarationExpression()` to compute some intermediate source range.

All of these modified elements are relevant to the change represented by transaction #2416. First, the developer removed the class `CopyPositionsMatcher` introduced in transaction #1476 and inlined it (with some modifications) in the `ASTConverter.setTypeForVariableDeclarationExpression()`. Then, the developer modified some code to compute the source range in a way that is highly similar to how the code introduced in transaction #1470 worked.

In summary, the modifications performed in transaction #2416 involved understanding the code changed in transactions #1470 and #1476 and using (by copying or inlining) some code introduced in these two transactions. However, the commit messages and the bug reports of these two transactions did not seem related at all, preventing their retrieval using standard search techniques. The clustering technique we describe in the rest of this section is able to identify the two related transactions given many combinations of pairs of methods in transaction #2416. In our scenario, having identified two relevant methods, the developer could have instantly obtained a *recommendation* consisting of the two transactions in the matching change cluster. With tool support such as SemDiff [13], the developer would then be able to seamlessly peruse the transactions to view any of the 21 elements changed, the log messages, or even the detailed changes to source code committed as part of the transactions in the cluster.

2.3. Overview of the technique

We model a program $P = \{e_1, \ldots, e_n\}$ 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 $C \subseteq P$[‡]. 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

[‡]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.

DOI: 10.1002/smr

related to the task. Given a query, our technique retrieves the relevant clusters in four steps:

1. *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*.
2. *Cluster transactions*: The clusterable transactions are clustered based on the number of overlapping elements using the nearest-neighbor clustering algorithm defined in Figure 1. This algorithm assumes that program elements (e.g., fields or methods in Java) can be uniquely represented, and 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. 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. The result of this clustering operation is a set of clusters representing transactions that involve 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 change history of systems not used in the evaluation revealed four elements as the threshold leading to the most balanced results [14]. 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 [11], 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.
3. *Retrieve the clusters matching a query*: We return all the clusters $C \in \mathcal{C} | Q \subseteq C$.
4. *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 preliminary studies [11,14]. 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 any transaction in the past.

We define the function [highFrequency$(e, T) \rightarrow$ boolean] as returning *true* if element e occurs in 3% or more of the transactions in T. We designed this heuristic through prior experimentation

DOI: 10.1002/smr

```
1: Input: T = {T_1, ..., T_n}: A sequence of transactions
2: Parameter: MinOverlap: A positive, non-zero value indicating the minimum overlap
   between two transactions in a cluster
3: Var: C: A set of clusters, initially empty.
4: for all T_i ∈ T do
5:    MaxOverlap ← 0
6:    MaxIndex ← −1
7:    for all C_j ∈ C do
8:       if |C_j ∩ T_i| > MaxOverlap then
9:          MaxOverlap ← |C_j ∩ T_i|
10:         MaxIndex ← j
11:      end if
12:   end for
13:   if (MaxIndex >= 0) ∧
         (MaxOverlap ≥ MinOverlap) then
14:      C_MaxIndex ← (C_MaxIndex ∪ T_i)
15:   else
16:      NewCluster ← T_i
17:      C ← C ∪ { NewCluster }
18:   end if
19: end for
20: return C
```

Figure 1. Nearest-neighbor clustering algorithm.

after noticing many situations where some central or critical elements were continually modified. 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 clustered transactions 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 initial prototyping 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.

DOI: 10.1002/smr

Heuristic 4 (*Maximum time span*). We reject any cluster whose last (most recent) transaction is more than a threshold t days earlier than the date of the query.

For example, with a 30-day threshold, for a query performed on July 1, 2008, we would reject a cluster whose last transaction was performed on April 15, 2008. This heuristic was introduced after studying the results reported in our previous experiment [14], where we noticed that many reported clusters were not useful simply because the reported elements were stale, a well-known risk when analyzing historical data [15, p. 125]. Based on a preliminary study of our previous results, and our experience with the study of software evolution in general, we opted to study three thresholds for this heuristic: 30, 90, and 180 days.

3. EMPIRICAL ASSESSMENT

The overall goal of this research was to assess *to what extent can we use past changes to inform software investigation?* This section describes the empirical setup we designed to answer that question. We start by refining our general research goal with three specific research questions.

3.1. Research questions

Q1. Support frequency. What percentage of change tasks relate to change 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 use to produce recommendations to support ongoing maintenance.

Q2. Impact of Heuristics. What is the impact of the filtering heuristics on the overall precision of the recommendations?

Q3. Value of the information. To what degree are recommended change 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 (to filter out large code cleanups and major code reorganizations unlikely to map to focused changed tasks). 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 600 analyzable

DOI: 10.1002/smr

transactions in the system's change history. We analyzed 600 transactions because this was the largest common 100-multiple of analyzable transactions among all the target systems.

3. For each *analyzable* transaction T_i (for $i > 200$), we apply the clustering algorithm of Figure 1 to the set of all *clusterable* transactions $\{T_j \mid j < i\}$.
4. For transaction T_i, we produce a set of queries $\mathcal{Q} = \{(e_m, e_n) \mid (e_m, e_n) \in T_i \times T_i \wedge e_m \neq e_n \wedge \{e_m, e_n\} \cap \text{additions}(T_i) = \emptyset\}$. In other words, we consider all possible combinations of two elements in T_i that do not correspond to elements added as part of the transaction. Although, in theory, $|\mathcal{Q}| \approx \binom{|T_i|}{2}$ can grow very large, 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 the possible starting points of two elements.
5. We retrieve the clusters formed by at least two 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 600 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 that produce at least one recommendation, the ratio of recommendations to the total number of queries for the transaction.

For instance, a transaction with five elements would generate $\binom{5}{2} = 10$ queries. If only four 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 retrieved change clusters 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(s), and hence the probability to produce a recommendation is 1.0. 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 three different classes and two different packages, generated from transactions that span at least 7 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.

DOI: 10.1002/smr

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. *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 to interpret the results for the purpose of assessing Q3.

Additional explanations of our experimental method, including an example application, can be found in a separate publication [14, Section 3.3].

3.3. Comparison with prior experiments

The methodology described in this section was used to *replicate* the experiment described in a previous paper [14]. We replicated the experiment because our initial results indicated a number of necessary improvements and additional trends worthy of being studied. Specifically, the experiment described in this paper replicates the previous experiment with the following differences:

- We have improved our transaction detection engine to automatically filter out any transactions where the only changes detected were in the comments to elements (as opposed to the code). This improvement is in response to the observation that many useless clusters had been detected for cleanups to the Javadocs comments.
- We increased the number of analyzed transactions considered from 500 to 600 (the largest possible 100-multiple given our available data).
- We investigated the effect of an additional heuristics, MAX TIME SPAN, with three different values.

As such, the specific goal of the experiment was to confirm the earlier findings with more rigorous analyses, obtain more accurate answers to our questions, as well as test the new heuristics.

DOI: 10.1002/smr

4. QUANTITATIVE RESULTS

We describe our data sources and quantitatively analyze the results of the experiment.

4.1. Data sources

Systems were selected among the population of available and long-lived open-source Java projects. We selected open-source projects that are sponsored and developed by different organizations, each with their own rules and development processes. However, these are not necessarily representative of all systems and, in particular, proprietary software products are likely to exhibit a different process. To be selected for analysis, a system needed to have at least 800 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 I provides the references to the systems we analyzed, and Table II reports on their main characteristics as related to our research questions. For each system, the columns *First* and *Last* give the dates of the first and the last analyzed transactions, respectively. These correspond to the 201st and 800th analyzable transactions available in the repository. The following column gives the number of days between the first and the last transactions analyzed. Column *Trans.* gives the total number of transactions committed between the first and the last analyzed transactions. For instance, for the Ant project, we analyzed 600 transactions out of 3853 because $3853-600=3253$ transactions were too 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 the number of clusters grows linearly with the number of transactions analyzed.

4.2. Q1: support frequency

Table III 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 600) and the corresponding ratio, respectively. The fourth column gives the input coverage measure.

Table II. Characteristics of target systems.

System	First	Last	Time	Trans.	Clusters	Pool	Ratio
Ant	6 Dec 2001	17 Jul 2007	2048	3853	1142	1231	1.078
Azureus	12 Nov 2003	14 Jul 2004	244	3103	851	960	1.128
Hibernate	4 Dec 2003	19 Aug 2005	623	3922	1038	1116	1.075
JDT-Core	17 Jan 2002	15 Jul 2003	544	4192	652	806	1.236
JDT-UI	20 Aug 2001	15 May 2002	268	3081	894	962	1.076
Spring	1 Feb 2004	6 Feb 2006	370	3627	1203	1303	1.083
Xerces	17 May 2001	8 Nov 2007	2366	2681	802	922	1.150
Total			6463	24459			

DOI: 10.1002/smr

Table III. Experimental measures—default configuration.

System	Supported	Ratio	Coverage	Scattered	Accuracy
Ant	87	0.15	0.17	0.29	0.23
Azureus	131	0.22	0.26	0.48	0.29
Hibernate	60	0.10	0.23	0.33	0.35
JDT-Core	189	0.32	0.20	0.74	0.24
JDT-UI	63	0.11	0.22	0.62	0.28
Spring	62	0.10	0.27	0.66	0.36
Xerces	148	0.25	0.31	0.64	0.28
Average	105.7	0.176	0.236	0.537	0.291

As this data shows, on average less than 1 in 5 analyzed transaction overlaps with change clusters. As it is expected, the number of supported transactions varies per system ($\sigma = 0.085$ for the ratio). The input coverage values show that on average close to 1 in 4 queries will yield a recommended cluster if a cluster can be recommended. Values of the input coverage metric are more stable ($\sigma = 0.046$).

The general answer to our first research question is that it is reasonable to expect that less than 1 in 5 maintenance tasks overlaps with change clusters. In addition, querying for change clusters based on two relevant methods will identify the cluster about 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 $0.176 \times 0.236 \approx 4\%$ of the time. This corroborates our previous estimate of 5%. The lower value with the replicated experiment is easily explained as an effect of our exclusion of comments-only modifications when determining transactions.

4.3. Q2: effect of filtering heuristics

To study the effect of the various heuristics, we calculated the effect of applying each 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). Finally, the heuristics were designed to improve the results; hence, our theory was that the heuristics should result in an increased accuracy, and we used our experiments to test this hypothesis using a one-tailed Wilcoxon signed-rank test. In the following, we consider results to be statistically significant at $p = 0.05$.

Table III also reports on the measure of *Supported Transactions*, *Scattered Clusters*, and *Accuracy* for the default configuration. In this configuration, Heuristics 1 (IGNORE HIGH FREQUENCY), 2 (MIN COHESION 0.6), and 3 (MAX TIME SPAN) 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. Compared with our previous experiment, the average value of *Supported* changed from 0.49 to 0.54, and the *Accuracy* changed from 0.33 to 0.29. Given the amount of natural variability in the different data sets we analyzed, we do not consider these changes to be significant.

DOI: 10.1002/smr

Table IV. Effect of filtering heuristics on the number of supported transactions (Sup.), the number of scattered clusters (Sca.), and the accuracy (Acc.).

	High frequency			Min cohesion			Min transactions		
System	Sup. (%)	Sca. (%)	Acc. (%)	Sup. (%)	Sca. (%)	Acc. (%)	Sup. (%)	Sca. (%)	Acc. (%)
Ant	−7	+3	+6	−39	−63	+29%	−71	−17	−71
Azureus	−13	0	+4	−63	−62	+49	−53	+58	−14
Hibernate	−13	−31	+6	−53	−46	+6	−52	+3	+12
JDT-Core	−2	+1	0	−47	−42	+25	−37	+17	−11
JDT-UI	0	+0	0	−52	−19	+20	−73	+33	+9
Spring	0	+0	0	−39	−5	+26	−68	+29	−55
Xerces	−20	−13	+13	−48	−43	+6	−40	+27	−0
Average	−7.9	−5.7	*+4.2*	−48.7	*−41.0*	*+23.0*	−56.1	*+21.4*	−18.7

Average numbers in italics indicate results significant at the 0.05 level.

Table IV shows the effect of the filtering heuristics on the variables of interest. The first column group provides the results of applying the IGNORE HIGH FREQUENCY heuristic. Applying this heuristic results in an average of 7.9% 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 ($p = 0.0488$). These overall results confirm our preliminary results [14].

The second column group of Table IV shows the effect of the MINIMUM COHESION (0.6) heuristic. Applying this heuristic results in an overall 48.7% 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 also a statistically significant positive impact on the accuracy. The negative effect on the number of scattered clusters is easily explained by the design of this heuristic. If we assume that most transactions tend to modify elements in the same classes, the requiring a minimum cohesion will eliminate overly scattered clusters, which may not be desirable and somewhat counterbalances the increase in the accuracy. These new results confirm the ones previously obtained but the trends are much more accentuated. The statistically significant impact on the accuracy had not been originally supported by the original experiment.

The third column group of Table IV shows the effect of the MINIMUM TRANSACTIONS (3) heuristic. Applying this heuristic results in an average of 56.1% 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 ($p = 0.0469$). 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 the accuracy also means that it might not be universally desirable. Although the per-system values are quite different from the original experiment (due to the large relative increase in the transactions analyzed given the filtering ratio), the overall trends are identical.

Table V shows the effect of the MAXIMUM TIME SPAN heuristic with values 180, 90, and 30, respectively. Applying this heuristic results in an average drop of 21.2 to 52.7%, depending on the threshold. There is no statistically significant positive impact on the number of scattered clusters reported. However, with a value of 180 the heuristic produces a significant increase in the accuracy

DOI: 10.1002/smr

Table V. Effect of Max Time Span 180,90,30.

System	Supported (180,90,30) (%)			Scattered (180,90,30) (%)			Accuracy (180,90,30) (%)		
Ant	−69	−82	−91	−36	−13	−13	+50	+85	+99
Azureus	−4	−18	−34	+4	+6	−8	+3	+11	+23
Hibernate	−10	−23	−45	−11	−9	−18	+2	−4	−1
JDT-Core	−15	−33	−56	+2	+3	+7	+4	+9	+21
JDT-UI	−6	−13	−33	+4	+9	0	+1	+3	+7
Spring	−16	−29	−58	−10	−11	−19	+8	+16	+10
Xerces	−28	−40	−52	−2	+2	+6	0	−23	−18
Average	−21.2	−33.9	−52.7	−6.8	−2.0	−6.4	*+9.8*	+13.8	+20.2

Average numbers in italics indicate results significant at the 0.05 level.

Table VI. Overlap.

System	Def. (%)	HF (%)	MC (%)	MTS (%)
Ant	49	53	57	56
Azureus	52	58	41	52
Hibernate	57	60	75	61
JDT-Core	56	57	51	58
JDT-UI	60	60	57	64
Spring	66	66	68	65
Xerces	61	65	47	65
Average	57	60	57	60

($p = 0.0156$). Applying the heuristic with the two other thresholds also led to positive average increases, but for thresholds of 90 and 30 days, the trends were not statistically significant at the 0.05 level.

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). Supported transactions with an average F-measure greater than zero. Table VI presents the results for the default configuration (Def.) as well as for the three heuristics that have a significant effect on quality: IGNORE HIGH FREQUENCY (HF), MIN COHESION 0.6 (MC), and MAX TIME SPAN 180 (MTS). The last row shows the (unweighted) average of averages (so as not to correlate the overall average with that of the systems with the most recommendations).

We observe that, on average, the recommended cluster for roughly 60% 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 change clusters. Combining this ratio with the average support ratio in Table III,

DOI: 10.1002/smr

we determine that we cannot reasonably expect that more than $0.176 \times 60\% = 11\%$ 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. In Section 5, we present the results of a detailed qualitative study that provides much deeper insights into the usefulness of the recommended clusters.

4.5. Threats to validity

In our overall evaluation of recommendation accuracy, we have 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. The likely impact of these three assumptions is that our results may underestimate the true potential of the approach. The analysis described in the next section sheds light on the impact of the first two assumptions.

5. QUALITATIVE STUDY OF RECOMMENDATIONS

The results we reported in the last section provide a precise and objective assessment of the degree to which tasks overlapped with change clusters for seven different systems. However, this assessment contributes limited insight into the potential usefulness of the clusters retrieved. We completed our investigation by conducting a detailed qualitative study of the clusters that would have been recommended to developers for the systems we analyzed. The goal of this qualitative study was to answer our third research question (Section 3.1) by determining *if recommended clusters would have been useful to developers*, and *why or why not*.

5.1. Methodology

The strategy for our qualitative analysis was to manually inspect all of the recommendations produced with the technique described in Section 3, and to assess whether, and why, the recommended clusters would have been useful (or not) in the given context. We chose to manually inspect the results after realizing the limitations of automated assessments based on the quantitative characteristics of the results, such as degree of overlap, or the characteristics of the clusters. Indeed, every development context is different and we felt that only a close inspection of each recommendation scenario would truly enable us to comment on the usefulness.

We therefore extracted the details of all of the 340 recommended clusters (across all seven systems) produced with the three valid heuristics enabled (IGNORE HIGH FREQUENCY, MIN COHESION 0.6, and MAX TIME SPAN 180). The analysis then proceeded in three phases. In the

DOI: 10.1002/smr

first phase, one of the two authors manually inspected the details of each recommendation. These details included all the change information for both the queried transaction and all of the transactions in the cluster. This information included, for all transactions: the commit message, the date of the transaction, the user name of the committer, the list of elements changed as part of the transaction, and, for the transactions in the cluster, the list of elements that overlapped with the queried transaction. From this information we could also derive the precision and recall of each transaction in the cluster with respect to the queried transaction.

Each of the authors inspected roughly half of the supported transactions. For each transaction, we created a set of notes that indicated whether the cluster would have been useful or not, with detailed justification.

In a second phase, both authors, as a team, reviewed and discussed the commented recommendations, and tried to group the descriptions into categories, following a process of *open coding* [16]. In addition to the information above, for this phase we used the SemDiff change-history analysis tool [13], which displays the detailed changes to the source code captured by any transaction. For example, using SemDiff, it was possible to determine whether a change to a method involved extensive rework or simply involved minor cosmetic changes such as renaming local variables. During this review phase, assessments were challenged by one of the authors and any question about the interpretation of the recommendation was answered through detailed analysis of the code changes. The result of this phase was the assignment of an *interpretation code* to each recommended cluster, which reflected the consensual assessment of both authors. As part of this process, all recommendations with an accuracy of 0 (i.e., for which there was no overlap of more than two elements between the cluster and the queried transaction) were coded as 'no match', and conservatively judged as not useful. The second phase elicited 17 distinct interpretation codes.

In the third phase, we revisited the data to interpret the meaning of the codes in relationship to each other. This phase resulted in the merging of different codes into a final set of 10 codes, and a conceptual organization of the relationships between different groups of codes.

5.2. Interpretation codes

Our qualitative analysis allowed us to distinguish between 10 clearly distinct situations representing the interaction between recommended clusters and a queried transaction (referred to as change task, or task).

No Match: This code was automatically assigned to any cluster that did not overlap with the task beyond an initial query. Although there is technically a small possibility that NO MATCH clusters could be helpful, we conservatively ignored this possibility and considered that NO MATCH clusters would not be useful. The NO MATCH code is the only one that was assigned without manual interpretation of a recommendation.

Spurious: This code was assigned to cases where we could not discover any semantic relationship among the transactions in the cluster, and between the cluster and the task. SPURIOUS cases thus represent accidental clustering of transactions that have nothing to do with one another, and therefore have no potential to help developers. For example, in some cases, spurious clusters were recommended because the queried transaction overlapped with a cluster that included a transaction corresponding to many changes to a large hub class.

DOI: 10.1002/smr

Management: This code represents cases where the *task* was related to code management and would have been very unlikely to benefit from change clusters. Examples of code management tasks include: modifications to comments, code cleanups, simple refactorings, trivial reverts to prior code, addition of testing code, etc. If a code management task was associated with multiple recommendations, we assigned the MANAGEMENT code to all recommendations.

Obvious: This code was assigned to cases where the cluster clearly represented a semantically cohesive set of elements associated with a high-level design concern, but where the elements in the cluster would have been totally obvious for a developer engaged in the associated task. For example, this code captured situations where the elements changed in the task are a proper subset of the set of elements changed in at least one transaction in the cluster.

Same Task: This code was assigned when the transactions in the cluster clearly pertain to the same general task as the queried transaction. This case is easy to observe because the transactions in the cluster and the queried transaction form a sequence in a short amount of time (1 or 2 days), by the same developer (or, very rarely, by a pair of developers).

Systematic: This code represents cases where the *task* was related to some systematic change and would have been very unlikely to benefit from history-based clusters. A systematic change is a change that naturally aligns with the structure of the code, such as changing all versions of an overloaded method, all implementors of an interface, etc. If a systematic task was associated with multiple recommendations, we assigned the SYSTEMATIC code to all recommendations.

Association: This code represents the case of a task that overlaps with the cluster because of an association rule, namely, there exists two or three elements that systematically change together. This case typically results in very low precision recommendations, where the only overlap between the cluster and the task that we observed were the elements instantiating the association rule.

Valuable: This code represents cases where there is a strong conceptual relationship among the transactions in the cluster, and between the cluster and the task, and the information in the cluster would not have been obvious to the developer because it spanned multiple classes and represents changes that did not occur recently, or that were not committed by the same developer responsible for the task. For example, the situation described in Section 2.2 corresponds to an actual case of VALUABLE cluster detected during our study. This code also includes cases where the task is a complicated revert to a previous state that is represented by the recommended clusters. We judged that these recommended clusters could be useful to help the developer understand the design they are reverting to. Because this code represents a positive recommendation, we assigned this code conservatively, if no other code could better describe the case. We also did not use fixed thresholds to make our decision, because our prior experience [14] showed that this approach was unsatisfactory for producing meaningful interpretations due to the large variety of programming styles and software change contexts.

Finally, we also created two codes to represent cases where transactions in the same cluster had different interpretations. For example, the VALUABLE/SPURIOUS code represents cases where one (or more) transaction in the cluster is clearly useful to know about whereas other ones appear to be spuriously associated with the task.

5.3. Results

Figure 2 provides a guide to interpreting our classification codes, and is the result of our manual analysis and classification of the recommendations. In the figure, the single (vertical) axis indicates

DOI: 10.1002/smr

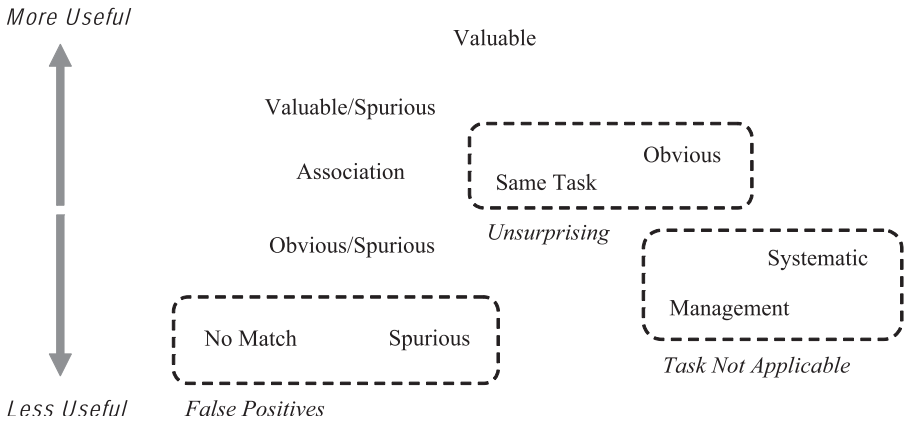


Figure 2. Interpreting the analysis codes.

the degree of estimated usefulness for a recommendation associated with a code. The figure also shows an additional level of classification for some of the codes. The group labeled *False Positives* captures cases where the clustering of transactions did not produce the intended results. Such cases are at the bottom of the figure as they constitute recommendations of negative value (that is, a developer would have to spend effort ruling them out as useful). The group labeled *Task Not Applicable* includes two codes that describe situations where recommendations would not have been very useful because of the characteristics of the task, as opposed to the characteristics of the recommendations. Within this group, we rate SYSTEMATIC cases as slightly more useful because we observed some systematic change tasks where additional information about prior work could have confirmed a developer in their decision to refactor some code. A third group, labeled *Unsurprising*, collects the codes representing situations where valid information is recommended, but this information would have been unlikely to add to the knowledge of a developer. We put this group at the boundary between useful and useless because, again, in a few cases the obvious information could have played a confirmatory role. Within this group, SAME TASK situations are the less likely to be useful given the short time span between recommended transactions and the corresponding task. Of the four remaining codes, OBVIOUS/SPURIOUS is ranked below the *Unsurprising* group because of the contamination from the spurious transaction; VALUABLE is ranked at the top by definition; VALUABLE/SPURIOUS is ranked just below VALUABLE for the value of the part of the cluster that is not spurious; ASSOCIATION is ranked relatively low because of the large number of false positives we observed in clusters that included an association rule.

Table VII reports on the number of codes assigned to recommendations in each of our systems. If we remove the 68 cases (20%) of recommendations produced for non-applicable tasks, we conclude that retrieving change clusters for our seven target systems produced 65% of false positives, 15% of valid but unsurprising information, and 13% of valuable information. Despite the fact that our clustering approach was not specifically designed to recover the association rules between elements, we were nevertheless surprised to discover only three clear cases of association rules for the 4200 transactions that we analyzed. This result appears to reinforce prior observations by Zimmermann *et al.* [3] that, although association rules are not uncommon between classes, such relationships

DOI: 10.1002/smr

Table VII. Results of the qualitative analysis.

Code	Ant	Azu.	Hib.	JCore	JUI	Spr.	Xer.	Avg. (%)
NO MATCH	6	17	4	30	9	12	25	30
SPURIOUS	2	7	3	32	6	3	22	22
MANAGEMENT	3	3	13	17	6	9	7	17
VALUABLE	1	5	0	14	4	4	8	11
OBVIOUS	2	5	5	10	1	0	3	8
SAME TASK	2	4	1	5	1	2	0	4
SYSTEMATIC	0	0	3	2	0	5	0	3
VALUABLE/SPURIOUS	1	0	0	4	1	0	1	2
OBVIOUS/SPURIOUS	0	0	0	4	0	2	1	2
ASSOCIATION	0	1	0	1	1	0	0	1

are much rarer at the level of individual methods. Although clustering classes or source files might have returned more recommendations, we focused on recommending individual methods because it is not clear how recommending large code elements like classes or files can support software investigation.

Considering our entire body of experimental work, we observe that in the seven systems we analyzed, change tasks are only rarely associated with change clusters. With our three valid filtering heuristics enabled, analyzing 4200 transactions resulted in the recommendation of only 340 change clusters. Of these change clusters, our qualitative inspection revealed only 36 valuable recommendations, for a total of 178 clear false positives. Putting it all together, we conclude that a simple overlap-based change clustering technique would have produced valuable recommendations in less than 1% of the tasks corresponding to individual commits to a revision control system. The implications of this result is that change clusters should only be recommended if it is possible to do so at close to no cost to the developers.

5.4. Threats to validity

The main threats to the validity of our qualitative analysis are investigator bias during the coding phase and limited external validity due to the system-specific characteristics of change streams.

A qualitative analysis involving manual inspection of the recommendations was necessary to obtain the rich interpretation reported in this section. However, this detailed interpretation is subjective and influenced by the experience of the authors. Notwithstanding the inherent subjectivity of the process, many factors contribute to the robustness of the classification. First, neither of the authors was involved in the development of any of the target systems, and as such our interpretation is likely to be similar to the intended audience for the recommendations (namely, developers unfamiliar with a system). Second, the classification was done in three phases, including first an individual coding and then a consensual assessment of the codes. Third, the categories emerged from the data, and were not initially planned by the investigators. Fourth, except in a few cases, the richness of the data we analyzed made the classification unambiguous, and for any situation that could be classified in more than one way, we automatically opted for the broadest and least useful code. Fifth, the purpose of our investigation was not to validate a specific recommendation technique, but rather to understand the potential of change clusters to help code investigation in general. Finally, our

DOI: 10.1002/smr

complete list of recommendations, associated tasks, and coding data is available upon request for independent analysis.

Analyzing the recommendation and task data in detail clearly illustrated the extent of the cultural differences in software change practices between projects. In some projects, change logs include detailed comments and represent coherent tasks. In other cases, most of the commit comments are empty and tasks are committed as series of short bursts. This general phenomenon is perhaps best evidenced by Table VII, which shows large discrepancies between projects. For example, the change history of Hibernate contained numerous code management tasks, whereas in the case of Azureus these were relatively rare. As a result, the interpretation of trends over change history can be very project specific. We were fortunate to select seven target systems that ended up covering a broad spectrum of change practices. However, the overall results we report are unlikely to reflect what can be expected from the change history of any specific system.

6. RELATED WORK

There exists the mining of software repositories [15,17] 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.

6.1. Mining software repositories

Our study was partially inspired by the work of Zimmermann *et al.* [3] and Ying *et al.* [10] 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 [18]. As opposed to the work cited above, Bouktif *et al.* used change patterns instead of association rules. In addition, 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 [19]. 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 [20]. 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.

6.2. 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 [21], possibly with human assistance [22]. Despite its long and rich history, experimentation

DOI: 10.1002/smr

with this approach continues to this day. For example, Andreopoulos *et al.* combined static and dynamic information [23], Kuhn *et al.* used a textual similarity measure as the clustering relation [24], and Christl *et al.* used clustering to assist iterative, semi-automated reverse engineering [25]. The main differences between most clustering-based reverse engineering techniques and the subject of our investigation are 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.* [9] and Vanya *et al.* [26] 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 their work and ours, but the purpose 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 investigation, 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 [27]. 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.

7. CONCLUSION

Developers often need to discover code that has been changed in the past. We investigated to what extent we can benefit from change clusters to guide program investigation. We defined change clusters as groups of elements that were part of transactions (or change sets) that had elements in common. Our quantitative analysis of over 17 years of software change data for a total of seven different open-source systems revealed that less than one in five tasks overlapped with a change cluster, but that even at this rate most of the recommended clusters are false positives. Applying a set of empirically-tested filtering heuristics on the data produced only 340 recommendations over the entire 4200 changes analyzed, a recommendation ratio of 8%. An in-depth qualitative analysis of the recommended clusters further showed that only 13% of the recommendation for applicable change tasks were likely to be useful. We conclude that recommending change clusters extracted from simple revision control systems amounts to finding needles in a haystack: valuable information exists but is well hidden. The ever-changing structure of the code, the lack of documented rationale for change sets, and the idiosyncrasies of the process used for committing changes, all hinder the discovery of relevant change sets. The practical implication of our findings is that change clusters should only be recommended if it is possible to do so at close to no cost to the developers.

DOI: 10.1002/smr

ACKNOWLEDGEMENTS

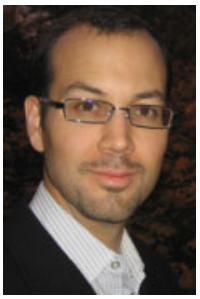
The authors thank Emily Hill and José Correa for their advice on the statistical tests, and the anonymous reviewers for their helpful suggestions. This work was supported by NSERC.

REFERENCES

1. Zeller A. The future of programming environments: Integration, synergy, and assistance. *Proceedings of the 29th International Conference on Software Engineering—The Future of Software Engineering*, 2007; 316–325.
2. Janzen D, Volder KD. Navigating and querying code without getting lost. *Proceedings of the 2nd International Conference on Aspect-oriented Software Development*, 2003; 178–187.
3. Zimmermann T, Weißgerber P, Diehl S, Zeller A. Mining version histories to guide software changes. *Proceedings of the 26th ACM/IEEE International Conference on Software Engineering*, 2004; 563–572.
4. Čubranić D, Murphy GC, Singer J, Booth KS. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering* 2005; **31**(6):446–465.
5. Wilde N, Scully MC. Software reconnaissance: Mapping program features to code. *Software Maintenance*: *Research and Practice* 1995; **7**:49–62.
6. Robillard MP. Topology analysis of software dependencies. *ACM Transactions on Software Engineering and Methodology* 2008; **17**(4):36. Available at: http://doi.acm.org/10.1145/13487689.13487691.
7. Marcus A, Sergeyev A, Rajlich V, Maletic J. An information retrieval approach to concept location in source code. *Proceedings of the 11th IEEE Working Conference on Reverse Engineering*, 2004; 214–223.
8. Sillito J, Murphy G, De Volder K. Questions programmers ask during software evolution tasks. *Proceedings of the 14th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2006; 23–34.
9. Kothari J, Denton T, Shokoufandeh A, Mancoridis S, Hassan AE. Studying the evolution of software systems using change clusters. *Proceedings of the 14th IEEE International Conference on Program Comprehension*, 2006; 46–55.
10. Ying ATT, Murphy GC, Ng R, Chu-Carroll MC. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering* 2004; **30**(9):574–586.
11. Robillard MP, Manggala P. Reusing program investigation knowledge for code understanding. *Proceedings of the 16th IEEE International Conference on Program Comprehension*, 2008; 202–211.
12. Zimmermann T, Weißgerber P. Preprocessing CVS data for fine-grained analysis. *Proceedings of the 1st International Workshop on Mining Software Repositories*, 2–6 May 2004.
13. Dagenais B, Robillard MP. Recommending adaptive changes for framework evolution. *Proceedings of the 30th ACM/IEEE International Conference on Software Engineering*, 2008; 481–490.
14. Robillard MP, Dagenais B. Retrieving task-related clusters from change history. *Proceedings of the 15th Working Conference on Reverse Engineering*, 2008; 17–26.
15. Kagdi H, Collard ML, Maletic JI. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution*: *Research and Practice* 2007; **19**(2):77.
16. Strauss AL, Corbin J. *Basics of Qualitative Research*: *Techniques and Procedures for Developing Grounded Theory* (2nd edn). Sage Publications: Beverley Hills, CA, 1998.
17. D'Ambros M, Gall HC, Lanza M, Pinzger M. Analyzing software repositories to understand software evolution. *Software Evolution*, Chapter 2. Springer: Berlin, 2008.
18. Bouktif S, Guéhéneuc Y-G, Antoniol G. Extracting change-patterns from cvs repositories. *Proceedings of the 13th Working Conference on Reverse Engineering*, 2006; 221–230.
19. Fluri B, Gall HC. Classifying change types for qualifying change couplings. *Proceedings of the 14th IEEE International Conference on Program Comprehension*, 2006; 35–45.
20. Breu S, Zimmermann T. Mining aspects from version history. *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, 2006; 221–230.
21. Hutchens DH, Basili VR. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering* 1985; **11**(8):749–757.
22. Sartipi K, Kontogiannis K. A user-assisted approach to component clustering. *Journal of Software Maintenance and Evolution*: *Research and Practice* 2003; **15**(4):265–295.
23. Andreopoulos B, An A, Tzerpos V, Wang X. Multiple layer clustering of large software systems. *Proceedings of the 12th Working Conference on Reverse Engineering*, 2005; 79–88.
24. Kuhn A, Ducasse S, Gîrba T. Enriching reverse engineering with semantic clustering. *Proceedings of the 12th Working Conference on Reverse Engineering*, 2005; 133–142.
25. Christl A, Koschke R, Storey M-A. Equipping the reflexion method with automated clustering. *Proceedings of the 12th Working Conference on Reverse Engineering*, 2005; 89–98.

26. Vanya A, Hofland L, Klusener S, van de Laar P, van Vliet H. Assessing software archives with evolutionary clusters. *Proceedings of the 16th IEEE International Conference on Program Comprehension*, 2008; 192–201.
27. Hassan AE, Holt RC. Replaying development history to assess the effectiveness of change propagation tools. *Empirical Software Engineering* 2006; **11**(3):335–367.

AUTHORS' BIOGRAPHIES



Martin P. Robillard is an associate professor in the School of Computer Science at McGill University in Montréal, Canada. His research focuses on techniques to decrease the cost of software evolution by reducing the knowledge and effort required by software developers involved in change tasks. He received his PhD in Computer Science from the University of British Columbia in 2004.

Barthélémy Dagenais is a PhD student in the School of Computer Science at McGill University in Montréal, Canada. His research focuses on the analysis of framework evolution and documentation to reduce the effort needed to use frameworks. He received his MSc in Computer Science from McGill University in 2008.

DOI: 10.1002/smr