Explaining why methods change together

Angela Lozano, Carlos Noguera, Viviane Jonckers
Software Languages Lab.
Vrije Universiteit Brussel
Email: {alozano,cnoguera,vejoncke}@vub.ac.be

***Abstract*—By analyzing historical information from Source Code Management systems, previous research has observed that certain methods tend to change together consistently. Co-change has been identified as a good predictor of the entities that are likely to be affected by a change, which ones might be missing modifications, and which ones might change in the future. However, existing co-change analysis provides no insight on *why methods consistently co-change*. Being able to identify the rationale that explains co-changes could allow to document and enforce design knowledge. This paper proposes an automatic approach to derive the reason behind a co-change. We define the reason of a (set) of co-changes as a set of properties common to the elements that co-change. We consider two kinds of properties: *structural properties* which indicate explicit dependencies, and *semantic properties* which reveal implicit dependencies. Then we attempt to identify the reasons behind single commits, as well as the reasons behind co-changes that repeatedly affect the same set of methods. These sets of methods are identified by clustering methods that tend to be modified in the same commit-transactions. We perform our analysis over the history of two open-source systems, analyzing nearly 19.000 methods and over 3700 commits. We show that it is possible to automatically extract explanations for co-changes, that the quality of such explanations improves when structural *and* semantic properties are taken into account, and when the methods analyzed co-change recurrently.**

I. Introduction

Co-change has been identified as an appropriate method to detect hidden dependencies [1], and as a good predictor of the impact of a change [2]–[4]. Nevertheless, co-changes have only been used to recommend future co-changes. In particular, extending co-change analysis to more informative recommendations (e.g., specific changes based on co-changing entities) has been unsuccessful [5] probably due to the large amount of coincidental changes [6]. Therefore, while co-change analysis has proven worthwhile, it only considers the frequency with which source code entities (co-)change, but no insight has been gained on *why they co-change*. Being able to derive the rationale behind co-changes *could allow* to document design knowledge, enforce design restrictions, and to make predictions for new methods.

This paper proposes an automatic approach to derive the reason behind a co-change.The granularity level for our approach is methods because they are functionality units with unique purposes, and therefore their relations may convey some underlying rationale. We define the rationale of a (set of) co-change(s) as a set of properties common to the elements that belong to the co-change, at that point in time.

The approach considers two kinds of properties that might provide reasonable explanations for a co-change. First, *structural properties* explain co-changes by evidencing explicit dependencies common to the elements that change. Examples of structural reasons include type references, exceptions thrown or cached, and the type that defines the method. Second, *semantic properties* explain co-changes by evidencing implicit dependencies common to the co-changing methods. Semantic properties include names of methods, parameters or local variables defined within the method, as well as recurring terms in the documentation (JavaDoc) attached to the co-changing methods. Our hypothesis is that it is necessary to mix structural and semantic properties to obtain reasons that: (1) cover a good percentage of co-changes, (2) are plausible explanations of the co-changes among those methods, (3) describe only the methods that should co-change, and (4) are different from each other –so that different co-change relations have different rationales. We propose a metric to evaluate each one of the previous characteristics, and we use them to evaluate the quality of different reasons. For instance, to evaluate the impact of different types of property, we compare the metrics for the reasons obtained with all properties *versus* those obtained using semantic or structural properties only.

Also, we compare the reasons found for two types of co-change relations. First, for sets of methods that co-changed in the *same commit transaction* and for sets of methods that *recurrently co-changed*. Given that several researchers have identified that commits tend to be either too fine grained or too coarse grained [6], [7] we expect to find that the reasons for methods co-changed in a commit transaction perform worse than the reasons for recurrently co-changing methods. This is because the latter ones are more likely to reveal logical dependencies like the ones behind methods implementing the same feature or concept.

We perform our analysis over the history of two Java open-source systems, analyzing nearly 19.000 methods and over 3700 (trunk) commits. We find that our approach successfully extracts reasons for co-changes among sets of methods; especially when the reasons are extracted from clusters of recurrently co-changing methods and use both semantic and structural properties. Also, that all approaches analyzed to extract reasons are acceptable in terms of coverage, uniqueness, and idiosyncrasy. This applies even for the most naive approach: single co-changes (per commit), and using one sort of properties (structural or semantic). Finally, we conclude that it is possible to automatically extract explanations for co-changes, however, more research is required in fine tuning the parameters for detecting clusters of recurrently co-changing methods. Reasons for clusters of recurring co-changes suffer from bad coverage, but are the ones with the most potential for prediction and documentation since they provide the most plausible explanations.

The rest of the paper is organized as follows: Section II

introduces the properties that used to describe methods, commits and clusters of commits. In this section we propose five research questions and explain the evaluation framework to answer them. Section III gives an overview of the data collection procedure and execution, before presenting our findings in Section IV. Section V presents a discussion of our findings, and Section VI positions them with respect to existing related work. Finally, Section VII and Section VIII respectively present the threats to validity and conclude the paper.

II. Reasons for change

In order to provide an explanation for why methods change together, we consider the manner in which change ripples across the code base. We base ourselves on the hypothesis that change ripples along dependency links. When implementing a change, all the source code elements affected by the change will be related in some manner, that is, they will belong to the concern that is being modified. For example, consider a library for managing a bar and an application relying on it. Within this library a method `isAdult()` checks whether a user is indeed an adult. A new version of the library modifies the signature of this method, to take into account the country in which this functionality is called. The change then will ripple through the called-by relations in the application. When looking at the commit transaction that accommodates to this change, all methods in the change will invoke the `isAdult` method. Furthermore, because of the nature of the change, they will also have a reference to the type that provides location services, and maybe mention the word "Adult" in their documentation. These common properties then become a *reason* for the change. Now, new methods introduced to the application with properties in common with this reason (invoke `isAdult`, have a reference to Location services, and mention "Adult" in their documentation), will probably need to be taken into account whenever the original group of methods change.

A. Property-based reasons

Our approach is based on using properties shared among co-changing methods to provide reasons for why they change together. We say that a description D_m of a method m is the set of properties for that method. Properties p are pairs $\langle k, v \rangle$ where k is the type of property, and v its value.

A reason R_M for a group of methods M is the properties that they have in common:

$$R_M := \bigcap_{m \in M} D_m$$

By choosing the group of methods in a particular commit transaction (M_c) we can assert the reason for that commit (R_{Mc}) as the intersection of the descriptions of the methods at the end of the commit. Thus, co-changing methods sharing at least one property when they co-change will have a non-empty reason: the common property.

Similarly, we define the reason for a set of consistently co-changing methods as the intersection of the reasons for each commit in which the methods co-changed.

Notice that, the *reason for a co-change* only makes sense if there are co-changing entities. Therefore, commit transactions consisting of a single method are marked as having an *undefined reason*.

We consider two sorts of properties, those that represent structural (or explicit) dependencies between methods, and those that represent semantic (or implicit) dependencies.

Structural	*Semantic*
1.CALLS_METHOD_NAME	1.LOCAL_VARIABLE_DECLARATION_NAME
2.CATCH_EXCEPTION_TYPE	2.METHOD_JAVADOC_MENTIONS
3.DECLARING_TYPE	3.METHOD_NAME
4.DECLARING_TYPE_EXTENDS	4.METHOD_PARAM_NAME
5.DECLARING_TYPE_IMPLEMENTS	
6.LOCAL_VARIABLE_DECLARATION_TYPE	
7.METHOD_ANNOTATED	
8.METHOD_PARAM_TYPE	
9.RETURN_TYPE	
10.THROWS_TYPE	

TABLE I. Properties analyzed to describe reasons for co-changes.

1) Structural properties: The structural properties selected in this paper describe calls-to relations, type-reference dependencies or scoping relations. Calls-to relations and type-reference dependencies might explain co-changes that occur because of coupling between entities: changes to called methods are likely to ripple out to their callers, and changes to shared types are likely to ripple to the users of those types. Scoping relations, on the other hand, might explain co-changes that occur because of cohesion within a module: methods within one class or hierarchy of classes might change together when the concept modeled by the class is updated.

For the calls-to relations we record for each method, the names of the methods invoked inside its body. For type-reference dependencies, we record the types of local variables defined inside of the method, the types of the method's arguments, the return and thrown exception types, as well as the types of annotations present in the method. Finally, we record the name of the method's declaring class, as well as the types it extends and interfaces it implements. Table III (first column) details the structural properties gathered.

2) Semantic properties: In addition to structural properties as possible reasons for why methods change together, we also include semantic properties (table III, second column). By including semantic properties we expect to capture implicit dependencies through which changes might ripple through a code-base. To do so, we take two sources of semantic information: the method's documentation, and the names used as identifiers within a method. For the method's documentation, we process the Javadoc comments from the method, remove stop words, and assert one property per word mentioned in the Javadoc. For the identifiers, we collect the names of local variables, parameters and the method's name.

Running example: Consider the commit transaction with message *'priority of task is save and open on xml files'* and time-stamp *'2003-05-22 22:56'* of GanttProject which modified two methods*. Using the properties

*GanttXMLSaver.writeTask(OutputStreamWriter,int,String) and GanttXMLOpen.DefaultTagHandler.startElement(String,String,String,Attributes

defined above, we describe the methods on this commit transaction using 93 properties (50 for the first one, and 43 for the second one). Since the methods shared 9 characteristics, the *reason for the commit transaction* becomes: CALLS_METHOD_NAME: add, equals, get, getLength, size, toString; LOCAL_VARIABLE_DECLARATION_NAME: i, task; and LOCAL_VARIABLE_DECLARATION_TYPE: GanttTask.

B. Some reasons are better than others

With this set of properties as possible explanations for co-changes, we aim at answering five research questions:

RQ1: To what extent is it possible to automatically find a reason for a set of co-changing methods? We analyze this question by comparing the coverage of the reasons found:

Coverage: This relates to the number of commit-transactions that have non-empty reasons. We define two types of coverage.

Coverage per commit, Cov_C as the ratio of commits with a non-empty C_R reason to total number of commits in the system's history C_s. And coverage per methods, Cov_M as the ratio of methods with a non-empty M_R reason[†] to total number of methods in the system's history M_s.

$$Cov_C = \frac{C_R}{C_s} \qquad , \qquad Cov_M = \frac{M_R}{M_s}$$

Given that we eliminate commits in which only one method changes, and that commits in which many methods change are unlikely to have a single reason, the coverage of our approach will be low.

RQ2: To what extent the automatically detected reasons describe only the set of co-changing methods? We analyze this question by assessing the discriminating power of reasons.

Idiosyncrasy: Good reasons will contain properties that tend to occur *only* in methods that change together. If the properties found in a reason are also found in methods that did not change together, then those properties are likely found by a coincidence and do not represent an explanation for the change.

Therefore, we measure the idiosyncrasy $Idios(R_M)$ of a reason R_M as one minus the ratio between the set of methods that are described by R_M by coincidence (i.e., the number of methods that have properties in common with R_M but that do not belong to the methods it describes $-M-$) and the total number of methods in the system's history M_s.

$$Idios(R_M) = 1 - \frac{|\cup_{m \in M_s} R_M \subset D_m| - |M|}{|M_s|}$$

For example the idiosyncrasy for the example commit is: 1-((5[‡] - 2[§])/(14895[¶])) = 1- 0.0002 = 0.9998.

[†]Methods modified in at least one commit with non-empty reasons.

[‡]Methods whose description includes all the properties of the example commit: GanttXMLSaver.writeTask(OutputStreamWriter,int,String) GanttXMLOpen.DefaultTagHandler.startElement(String,String,String,Attributes) GanttXMLOpen.GanttXMLParser.startElement(String,String,String,Attributes) GanttXMLSaver.writeTask(OutputStreamWriter,DefaultMutableTreeNode,String) GanttXMLSaver.writeTask(Writer,DefaultMutableTreeNode,String)

[§]Number of methods modified in the example commit see section II-A.

[¶]Methods belonging to GanttProject during the period analyzed (table III)

RQ3: To what extent the automatically detected reasons for co-changing methods overlap with each other? We analyze this question by measuring the uniqueness of reasons.

Uniqueness: It is also important to know whether reasons are sufficiently different between each other to serve as explanations *only* for the changes they describe. Thus, we measure the similarity $Sim(R_1, R_2)$ between two reasons as their Jaccard index (i.e., the intersection over the union of their properties.).

The uniqueness of a reason R_i is the mean difference to the rest of reasons found in the project (i.e., R).

$$Unq(R_i) = 1 - \tilde{x}(\bigcup_{R_j \in R \wedge i \neq j} Sim(R_i, R_j))$$

Uniqueness tells us if different co-change relations are likely to be due to different reasons. Lets suppose that there are three methods (m1, m2, and m3) but there are only two co-change clusters (m1 and m2, m2 and m3). Even though m2 co-changes with m1 and m3, it is likely that the reasons for co-changing with m2, are different from the reasons for co-changing with m3. Therefore we expect the reasons to be unique. For example the uniqueness for the example commit is[‖] 0.985333.

RQ4: To what extent the automatically detected reasons for sets of co-changing methods are sound? We analyze this question by manually checking their plausibility.

Plausibility: Even if reasons explain a large number of methods (high Cov_M), covering most of the history of the application (high Cov_C), being sufficiently discriminatory (high $Idios$, and high Unq), they might still make no sense. Therefore, we need to perform a manual assessment of the reasons found to see whether they are plausible. This is achieved by comparing the reasons for a (set) of commits produced by our analysis with the message that corresponds to the (set of) commits. We assert each reason as either plausible or not. We consider a reason a *plausible* explanation for a commit, if the words or terms mentioned in the properties of the reason appear in the commit message that accompanies the co-change. For groups of co-changes, for those that span more than 6 co-changes, we extract a word-cloud from the commit messages to provide an overview of the general terms therein. We apply a small degree of liberty when aligning the terms of the reason with those present in the commit message(s); for example if the commit message mentions UI, or mouse events, we will consider plausible reasons those that include dependencies to JPanel or ActionListener types (both types present in Java's SWING UI framework). Finally, note that the plausibility depends on the quality of the commit message, if the commit message provides no information (i.e., "bugfix" or "no message") we will consider the reason as an *implausible* explanation for the commit regardless of the properties found in the reason itself.

The example commit is considered plausible because the commit message and the reason refer to an I/O task[**].

[‖]The example commit had a non-empty intersection with 128 commits (from the 626 commits in GanttProject with a non-empty reason). Moreover, most of the intersections had only one property. Therefore, its similarity with other commits is very low (Min: 0, 1st Qu.:0, Median:0, Mean:0.018, 3rd Qu.:0, Max:0.3).

[**]See the description of the example in section II-A.

RQ5: Which is the best approach to identify the rationale of co-changes?
(a) Are the reasons extracted from recurrently co-changing methods better from those derived from single commits?
(b) Are the reasons that take into account structural *and* semantic properties significantly better from those that take into account only one type of properties?

We analyze these question by comparing the metrics presented above for reasons extracted from different properties and co-changes (see table II):

(a) the sets of methods that co-change in each commit transaction *versus* clusters of methods that regularly co-change

(b) all properties *versus* structural properties, and all properties *versus* semantic properties.

Notice that the range of all metrics proposed is between zero and one. The closer they are to one, the better are the reasons evaluated.

	All	Structural	Semantic
Commits	R_{CmAll}	R_{CmSt}	R_{CmSm}
Cluster	R_{ClAll}	R_{ClSt}	R_{ClSm}

TABLE II. NAMES FOR THE DATA SETS OF REASONS COLLECTED

III. DATA COLLECTION

This section explains which data was collected, and how it was processed to find the reasons for different co-changes. First of all, it is necessary to identify for each commit which methods were changed and their description at that point in time. This is done by:

1) Grouping commits of the trunk by author, message, and time (with a standard three minute sliding window).
2) For each commit, downloading a copy of the repository.
3) For each commit, identifying which methods were modified:
   a) For each source code file, retrieving the methods and their boundaries in terms of lines of code .
   b) Mapping the changes in the change log (i.e., lines changed, added or deleted per file) to the list of methods using their boundaries.
4) For each commit, describing each one of the modified methods:
   a) Getting the Abstract Syntax Tree per compilation unit (without type bindings) in the files that were modified.
   b) Gathering information over AST elements present on each method modified.

Once this data is collected (for each project) it is possible to find the reasons per commit and per cluster of recurrently co-changing methods, and to compare the results when analyzing all properties, *versus* the reasons that only contain structural properties or semantic properties.

1) Analysis per commits: Finding the reason for a commit is the result of intersecting the properties of the methods that change in that commit.

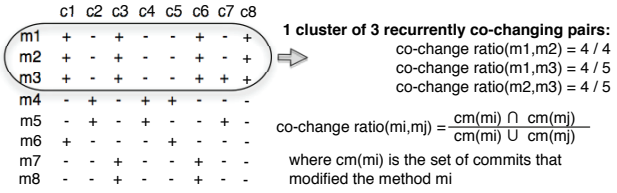

Fig. 1. Clusters of recurrently co-changing methods in a fictitious application.

2) Analysis per clusters of recurrently co-changing methods: Finding the cluster of methods requires detecting the set of methods that recurrently co-change, and intersecting their properties whenever they co-change. That is,

1) Finding pairs of frequently co-changing methods: that is sets of pairs that tend to be modified in the same commits (see Fig. 1):
   a) The methods in a pair can 'avoid' co-change in at most 20% of all commits in which any methods were modified. This means that methods pairs which changed in less than five commits, must co-change in all their commits to be considered as recurrently co-changing. In contrast, methods pairs which changed in five or more commits can have commits in which they did not co-changed.
   b) Having one or two co-changes could be due to coincidental co-changes. Therefore, method pairs must co-change at least three times to be considered a recurrently co-changing method pair.
2) Recovering the properties that appear in both methods of each pair of frequently co-changing methods (i.e., obtained in the first step) , for all of the co-changing commits
3) Clustering method pairs that co-changed in the same commits, and re-calculating their combined reason.

IV. EXPLAINING WHY METHODS CHANGE TOGETHER

We have collected structural and semantic properties from the historic information of two open-source projects, as found in their CVS repositories. The projects are GanttProject, a project scheduling and management tool and FreeCol, a turn-based strategy game[††]. We have selected these projects because of their ease of access to their repositories, and the fact that we could verify the data collected as we have analyzed them for other experiments [8]. Counting both projects, we process 3788 revisions and 18994 unique methods.

The structural information gathered for each method at each revision is summarised in Table III. For each method, we gather information regarding which methods are called (selector), type references in local variables, parameters, thrown and caught exceptions[‡‡]. Additionally, we record the method's name, name of parameters, and names of local variables declared in the method. Notice that in the table, we gathered

[††] ganttproject.cvs.sourceforge.net:/cvsroot/ganttproject, freecol.cvs.sourceforge.net:/cvsroot/freecol

[‡‡] References to primitive types (like int, float, boolean, etc.) and to java.lang.String are not taken into account since they are unlikely to be related to the underlying reason for a co-change.

the same number of values for METHOD_PARAM_TYPE and METHOD_PARAM_NAME, this is not the case for local variables, since Java allows several variables to be declared in the same statement (e.g, `int a,b,c`). For semantic properties, we processed the JavaDoc comments attached to the files, extracting a list of terms present in the comment.

Property	FreeCol	GanttProject
commits (with a non-empty reason)	1 087 (286)	2 701(626)
# clusters (methods involved)	7(14)	138(280)
# unique methods	4 099	14 895
# properties	478 312	54 7394
Structural properties		
CALLS_METHOD_NAME	202 874	251 118
CATCH_EXCEPTION_TYPE	1 987	2 684
DECLARING_TYPE	11 177	21 536
DECLARING_TYPE_EXTENDS	8 371	14 823
DECLARING_TYPE_IMPLEMENTS	6 427	13 426
LOCAL_VARIABLE_DECLARATION_TYPE	36 704	39 401
METHOD_ANNOTATED	2	422
METHOD_PARAM_TYPE	18 706	25 594
RETURN_TYPE	13 017	25 361
THROWS_TYPE	604	1 176
Semantic properties		
LOCAL_VARIABLE_DECLARATION_NAME	37 527	39 818
METHOD_JAVADOC_MENTIONS	108 985	60 568
METHOD_NAME	13 226	25 874
METHOD_PARAM_NAME	18 706	25 594

TABLE III. SUMMARY OF DATA MINED

A. Describing the reasons found

Intuitively, the number of methods is inversely proportional to the size of the reason. On one hand, the data-sets of reasons per commit tend to represent more methods, with more general reasons (i.e., reasons with few properties) as the top part of Fig. 2 shows. This was foreseeable given that when considering commit transactions, we eliminate from the analysis those commits in which only one method was changed. However, we do not perform special treatment of commits in which a large number of methods were changed, for example commits in which a large restructuring of the application was made. Because of the nature of the analysis, such large commits are unlikely of having a reason (that is, not a single property that is common to all changed methods), or having a very general reason (that is, a reason with one or two properties). For instance, the largest commit in GanttProject modified 3572 methods but they do not share any properties. While the largest commit *with a reason* changed 1177 methods; its reason contains only one property which is rather generic ('*DECLARING_TYPE_EXTENDS:GanttLanguage*'). In contrast, the reasons for small commits (i.e., that modified a couple of methods) can be very specific (i.e., with several properties), very generic (i.e., with few properties) or somewhere in between.

On the other hand, the data-sets of reasons per cluster provide more specific reasons (i.e., reasons with several properties) regardless of the number of methods described (see bottom part of Fig. 2). For Freecol, we find a very low number of clusters (7 in total) , all of them containing only two methods, but with a varied size of reason. For GanttProject, we find a much higher number of clusters. In general, GanttProject's clusters and commits follow an inverse relation between the number of methods they describe and the number of properties as Freecol's commits. In both data-sets, and for both commits and clusters, using all properties (marked as a circle in Fig 2) provides reasons with a high level of detail (size of reason) that also describe a larger number of methods.

B. RQ1: To what extent is it possible to automatically find a reason for a set of co-changing methods?

Given that the conditions to cluster co-changing methods are strict (at least 80% of co-change similarity and at least three changes) we expected that the reasons for commit transactions to cover more methods and more commits than the reasons for clusters of recurrently co-changing methods. Considering the properties taken into account, we expected that it would be more likely to find at least one common property for all co-changing methods when both structural and semantic properties are considered i.e., analyzing all properties would lead to having more reasons, and therefore to cover more methods or commits. Nevertheless, any difference between structural and semantic reasons would depend on the application analyzed and the quality of their comments and naming conventions. Table IV, which lists the coverage of methods (Cov_M) and commits (Cov_C) for both projects and all kinds of properties, confirms our expectations. Reasons extracted from single co-changes (i.e., commits) represent a higher percentage of the methods and of the commit transaction of the applications analyzed. Also, having both sorts of properties improves the coverage in comparison with reasons obtained from structural or semantic properties only. Finally, there is no consistent behavior when comparing the coverage of reasons obtained from structural properties against those obtained from semantic properties.

	R_{CmAll}	R_{CmSt}	R_{CmSm}	R_{ClAll}	R_{ClSt}	R_{ClSm}
Freecol						
Cov_M	0.17	0.14	0.11	0.0034	0.0029	0.0034
Cov_C	0.26	0.19	0.20	0.0064	0.0055	0.0064
Ganttproject						
Cov_M	0.29	0.27	0.06	0.015	0.016	0.011
Cov_C	0.25	0.23	0.13	0.053	0.049	0.044

TABLE IV. COVERAGE RESULTS

C. RQ2: To what extent the automatically detected reasons describe **only** the set of co-changing methods?

The usefulness of a reason for predicting co-changes depends on how well it is capable of pointing out *only* at the methods that are likely to change, that is having a high idiosyncrasy value. As Figure 3 shows, in general, all reasons extracted presented good idiosyncrasy values; the medians are above 0.981 for Freecol, and above 0.9972 for GanttProject. This indicates that when a reason for a co-change of methods is found, the reason represents less than 2% (for Freecol) and 1% (for GanttProject) of the methods in the application.

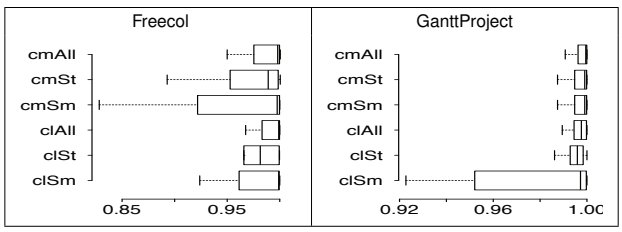

Fig. 3. Idiosyncrasy for the different data-sets for the applications analyzed.

When looking at commit-based reasons *versus* cluster-based reasons in terms of idiosyncrasy, fig. 3 shows that

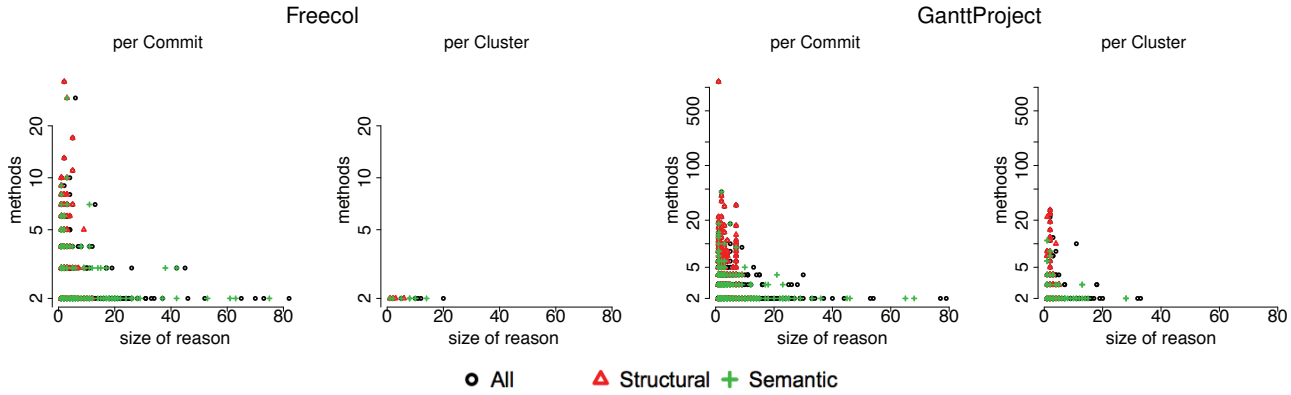

Fig. 2. Number of methods (y-axis -in logarithmic scale-) vs. Size of reason (x-axis) for the different data-sets of the applications analyzed.

commit-based reasons are better when compared against cluster-based reasons. Regarding at the type of properties used, having `all` properties tend to result in better idiosyncrasy values than having only `structural` or `semantic` properties. Finally, Figure 3 shows that `structural` properties tend to be worst when analyzing `cluster-based` reasons for both projects.

D. RQ3: To what extent the automatically detected reasons for co-changing methods overlap with each other?

Reasons should not be similar to each other, as they are supposed to describe sets of co-changing methods. Having similar reasons would indicate that they are not a good way to differentiate different sets of co-changing methods. Fig. 4 shows that, in general, the reasons we extracted tend to be unique (with a value above 85%). Similar to the idiosyncrasy results, commit-based reasons tend to have better uniqueness values for GanttProject while cluster-based reasons tend to have better uniqueness values Freecol. In terms of the type of properties used, reasons using only `semantic` properties are more likely to be unique than reasons using `all` or only `structural` properties (except for reasons extracted from commits in Freecol whose best uniqueness corresponds to `all`).

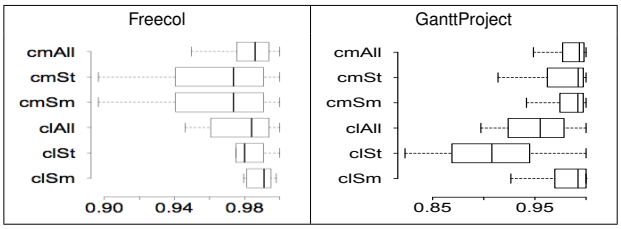

Fig. 4. Uniqueness for the different data-sets for the applications analyzed.

Finally, fig. 3 shows that `structural` properties tend to be the worst when analyzing `cluster-based` reasons for both projects.

We also expected that the level of detail of a reason to be directly proportional to its uniqueness. That is, reasons with a small set of properties would be more likely to overlap with other reasons. In general, the higher the size of a reason the more unique (see Fig. 5); with the majority of reasons having less than 20 properties while maintaining an uniqueness value above 93%. Therefore, in general it is not necessary to have many properties for a reason to be unique.

E. RQ4: To what extent the automatically detected reasons for sets of co-changing methods are sound?

Probably the most important evaluation criteria for reasons is to what extent the reasons that are extracted automatically indeed convey the rationale of co-changes or explain what is the logical dependency among the co-changing methods. Table V presents a couple of examples (taken from Freecol) of plausibility when analyzed per commits. The results on the

Reason	Commit description
Plausible	
DECLARING_TYPE: ColonyPanel, DECLARING_TYPE_EXTENDS: JLayeredPane, DECLARING_TYPE_IMPLEMENTS: ActionListener, METHOD_JAVADOC_MENTIONS: panel	changed cargo panel management
Implausible	
DECLARING_TYPE: Player, DECLARING_TYPE_EXTENDS: FreeColGameObject, METHOD_JAVADOC_MENTIONS: player	added initial contact support (when you first meet a nation/tribe)

TABLE V. EXAMPLES OF PLAUSIBILITY ANALYZED PER COMMITS

plausibility of the reasons extracted are shown in Table VI. The columns named 'Reasons analyzed' on Table VI indicate the amount of reasons whose plausibility was manually validated, while the columns name 'Plausibility' indicate the proportion of reasons found to be plausible. Given that this analysis is done manually, we performed an evaluation for a fraction of the reasons found (between parenthesis in the table). Results

	Freecol		Ganttproject	
Data-set	Reasons analyzed	Plausibility	Reasons analyzed	Plausibility
R_{CmAll}	57 (20%)	0.31	140 (20%)	0.26
R_{CmSt}	44 (20%)	0.31	126 (20%)	0.19
R_{CmSm}	45 (20%)	0.26	75(20%)	0.28
R_{ClAll}	7 (100%)	0.71	29 (20%)	0.58
R_{ClSt}	6 (100%)	0.33	27 (20%)	0.62
R_{ClSm}	7 (100%)	0.71	24 (20%)	0.33

TABLE VI. PLAUSIBILITY RESULTS (OF RANDOMLY CHOSEN REASONS)

indicate that reasons extracted from clusters are much more likely to describe logical dependencies among the co-changing

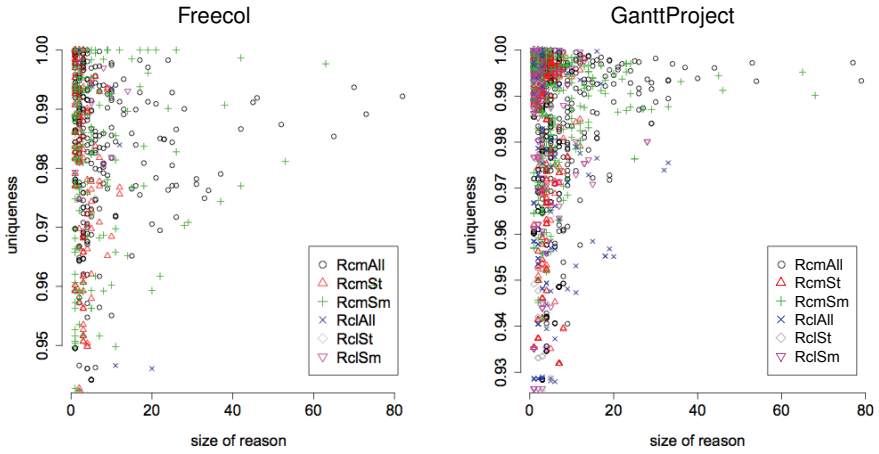

Fig. 5. Uniqueness vs. Size of reason for the different data-sets for the applications analyzed.

methods than those extracted from commit transactions. This result was expected as clusters represent sets of methods that recurrently co-change, and thus are more likely to have structural and semantic similarities. Regarding the type of properties used, it is not clear which properties are more likely to convey plausible reasons. For Freecol, merging semantic and structural properties increases the chance of finding plausible reasons regardless of the sets of methods analyzed, while for GanttProject semantic properties better describe commits and structural properties better describe clusters.

F. RQ5: Which is the best approach to identify the rationale of co-changes?

Finally, we want to compare which analysis would provide better reasons. We first consider the results that compare commits and clusters, for which we expect clusters to provide better results as they are less sensitive to coincidental co-changes. Then, we consider the results that compare different types of properties to extract the reasons. We expect that the combination of semantic and structural reasons would result into better reasons because they would have more properties that are orthogonal to each other. Finally, we discuss the most advantageous data sets.

(a) *Are the reasons extracted from recurrently co-changing methods better from those derived from single commits?:* Table VII shows the p-values for the statistical comparison of the idiosyncrasy and uniqueness values of reasons extracted from clusters *versus* those extracted from commits. Whenever the null hypothesis could be rejected, the table also shows what would be the relation between the metrics of cluster-based reasons *versus* those of commit-based reasons. The null hypotheses could not be rejected for the reasons extracted for Freecol except for uniqueness using semantic properties. This indicates that there is no difference in the idiosyncrasy and uniqueness of reasons extracted from Freecol regardless of the sets of methods analyzed. A possible explanation is that the non-empty reasons extracted from Freecol represent a smaller set of methods than those extracted from Ganttproject (see Fig. 2) resulting in more precise properties. However, when using only semantic properties on Freecol's code, cluster-based reasons are more unique than commit-based reasons. In contrast, the majority of null hypotheses were rejected for the reasons extracted from GanttProject. Cluster-based reasons taking into account all properties or only structural properties present better uniqueness and idiosyncrasy than commit-based reasons. Finally, commit-based reasons with only semantic properties are more unique for GanttProject than cluster-based reasons, contradicting the results found for Freecol.

Null Hypothesis: *Extracting reasons from* `clusters` *of recurrently co-changing methods using <prop> properties does not significantly improve <metric> results compared to extracting reasons from* `commit transactions`.

		Freecol		GanttProject	
<metric>	<prop>	H0	Alternative	H0	Alternative
Idios	all	0.69	none	8.2e-03	(>) 4.1e-03
	structural	0.93	none	3.9e-10	(>) 1.9e-10
	semantic	0.83	none	0.88	none
Uniq	all	0.69	none	8e-28	(>) 4.4e-28
	structural	0.47	none	8e-37	(>) 4.2e-37
	semantic	0.04	(>) 0.02	*1e-01	(<) 5.1e-02

TABLE VII. COMPARING REASONS EXTRACTED USING DIFFERENT SETS OF CO-CHANGING METHODS (WILCOXON TEST)

(b) *Are the reasons that take into account all properties significantly better from those that take into account only one type of properties?:* Table VIII shows the p-values of comparison for the idiosyncrasy and uniqueness values of reasons extracted with different sets of properties. Similarly to table VII, the table shows the direction of the relation whenever a difference between the data-sets could be established (i.e., the null hypotheses being rejected). Results indicate that taking into account all properties results in better uniqueness, and idiosyncrasy for commit-based reasons regardless of the application analysed. However, for cluster-based reasons, it seems that there is no clear difference in the uniqueness and idiosyncrasy when taking into account all properties *versus* structural or semantic properties. The lack of differences in the uniqueness for different types of properties could be explained by structural and semantic properties pointing out at the same source code relations.

(c) *What is the best configuration?:* Table IX shows the summary of the results obtained. The direction of comparing different data-sets, and the best value for each metric are taken into account to indicate which data-set resulted

Null Hypothesis: *Combining* `all` *properties to extract the reasons behind the co-changes of <set> does not significantly improve <metric> results compared to* `<prop>` *properties.*					
		Freecol		GanttProject	
<set>	<prop>	H0	Alternative	H0	Alternative
<metric>: Idios					
commits	structural	3.7e-06	(>) 1.8e-06	1.8e-03	(>) 9.4e-04
commits	semantic	0.20	none	1.8e-03	(>) 9.4e-04
clusters	structural	0.35	none	3.2e-04	(>) 1.6e-04
clusters	semantic	0.74	none	0.54	none
<metric>: Uniq					
commits	structural	2.9e-10	(>) 1.4e-10	1.3e-03	(>) 6.7e-04
commits	semantic	2.9e-10	(>) 1.4e-10	0.14	(>) 7.0e-02
clusters	structural	1	none	7.5e-14	(>) 3.7e-14
clusters	semantic	0.62	none	2.5e-15	(<) 1.2e-15

TABLE VIII. COMPARING REASONS EXTRACTED USING DIFFERENT PROPERTIES (WILCOXON TEST)

in better metrics. Commit-based reasons outperform cluster based reasons in coverage. However, it is not clear if commit and cluster based reasons have a different idiosyncrasy or uniqueness values. In any case, clusters are better to extract plausible reasons than commits (i.e., reasons extracted from single co-changes are more susceptible of lack of soundness). Considering different properties improves the quality of the reasons. It is more likely to describe only the methods changed in the same commit transaction by looking only at structural properties. In some cases, analyzing all properties does not improve the metrics obtained with a single type of properties. For instance, for the Cov_C, $Idios$, Unq, and $Plausibility$ of Freecol for cluster-based reasons with semantic properties only, or for the $Idios$ and Unq of Freecol for cluster-based reasons with structural properties only. This happens because the null hypotheses could not be rejected (results shown in table VIII) which in the case of Freecol could be due to the small size of the sample.

	Cov_M	Cov_C	$Idios$	Unq	$Plausib.$
which is better R_{CmAll} or R_{ClAll}?					
Freecol	Commit	Commit	None	None	Cluster
GanttProject	Commit	Commit	Commit	Commit	Cluster
which is better R_{CmSt} vs. R_{ClSt}?					
Freecol	Commit	Commit	None	None	Cluster*
GanttProject	Commit	Commit	Commit	Commit	Cluster*
which is better R_{CmSm} vs. R_{ClSm}?					
Freecol	Commit	Commit	None	Cluster	Cluster
GanttProject	Commit	Commit	None	Commit	Cluster
which is better R_{CmAll} vs. R_{CmSt}?					
Freecol	All	All	All	All	None
GanttProject	All	All	All	All	All
which is better R_{CmAll} vs. R_{CmSm}?					
Freecol	All	All	None	All	All
GanttProject	All	All	All	All	Semantic
which is better R_{ClAll} vs. R_{ClSt}?					
Freecol	All	All	None	None	All
GanttProject	Structural	All	All	All	Structural
which is better R_{ClAll} vs. R_{ClSm}?					
Freecol	All	None	None	None	None
GanttProject	None	All	None	Semantic	All

TABLE IX. COMPARISON OF METRICS FOR DIFFERENT DATA-SETS

V. DISCUSSION

This section discusses the variables that may have affected the results. The first variable is the parameters chosen for the clustering of the methods. Depending on the number of methods modified per commit of an application, having at least three co-changes might have been too strict a restriction. Similarly, if commits tend to modify too many methods it might be more difficult to find recurrently co-changing methods that indeed have logical dependencies. This is made evident by the number of clusters found in the applications analyzed. While the thresholds seem appropriate for GanttProject with 138 clusters, it is too strict for Freecol where only 7 clusters were found. Figure 6 shows the distribution of number of methods modified by commit, and changes made to unique methods for both applications. Methods in Freecol tend to change more, and its co-changes tend to be larger than those in GanttProject.Therefore it is more likely to find pairs of recurrently co-changing methods in Freecol. However, the coarser granularity of Freecol's commits may have reduced the chance of having a common reason among co-changing methods. Moreover, the higher chance of a method of being changed makes more difficult to find other methods with at least 80% of co-change ratio.

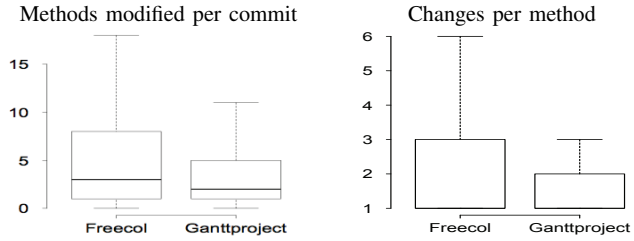

Fig. 6. Factors that may affect the effectiveness of the thresholds used for clustering co-changing methods.

Another variable that may have affected the results and the quality of the reasons found is the choice of properties. In terms of structural properties, the approach misses methods that are in the same call chain. Therefore, methods that co-change frequently because they call to each other might have empty reasons. This seems to happen frequently as shown by the coverage results. However, In terms of semantic properties, the approach taken to extract them is rather naive. The analysis of 'METHOD_JAVADOC_MENTIONS ' ignores the frequency of terms not only per method (i.e., a term that appears once in a single java-doc is regarded in the same way by the approach as one that appears several times) but also across methods (i.e., terms that are frequent across java-docs are not eliminated from the analysis). Ignoring the frequency of terms may have reduced the plausibility of the reasons by adding false-positive reasons (i.e., sets of methods that would otherwise have no reasons) by adding noise terms (i.e., making reasons more difficult to understand by adding properties that are not related to the dependencies among the set of co-changing methods e.g., 'returns') . Finally, although the words in identifiers (names of variables, parameters and methods) may point out to the underlying domain concept or feature being implemented, we did not perform any further processing (e.g., 'CamelCase' splitting or 'tolowercase' standardization).

VI. RELATED WORK

This section describes previous work on co-change analysis, and how they differ from this study.

A. Co-change analysis

Co-change analysis was initially proposed to detect logical dependencies that would have been too cumbersome to detect

using traditional static analyses [1]. Using more elaborate techniques (i.e., confidence and support of association rules), co-change analysis has shown its usefulness to detect incomplete changes at method level [4], and to identify the files affected by a modification task [9]. In fact, historical information has been found to be a better predictor of change propagation than structural dependencies [2]. Moreover, the prediction of future changes can be improved by adding the recency of changes to co-change analysis [3].

Co-change analysis has also been used to evaluate the modularization choices made, because it can uncover to what extent structural dependencies support developers in having changes hidden behind source code abstractions [8], [10]–[12]. For instance, finding architectural weaknesses and degradation by detecting discrepancies between change dependencies and structural dependencies [10], [11], providing evidence that modularization constructs that separate functionality from implementation (e.g., interfaces) facilitate software evolution by comparing the likelihood and size of changes of different modularization constructs [12], providing evidence that clones affect the maintainability of code by comparing the effort required to change methods with and without clones [8].

Given that co-change analysis does not depend on structural information it can be used to provide insights beyond source code artifacts, demonstrating its effectiveness for traceability detection [13].

B. Mixing structural vs. semantic information

Maletic and Marcus [14] were pioneers in merging structural and semantic data in reverse engineering and program comprehension. Their findings inspired recommendation systems that provide better results thanks to their combination of structural and semantic information [15], [16].

Kagdi et al. [17] combined semantic dependencies and change dependencies to increase the precision and when predicting software changes. Gethers et al. used Latent Semantic Indexing to find a baseline impact set for a change request, which was improved by adding co-change analysis and dynamic analysis [18].

C. Similarities and differences with previous work

We combine semantic and structural properties as they provide orthogonal views on the relations that source code entities may have because they tend to outperform approaches based only on the type of information (structural or semantic) especially for approaches aimed at improving understandability. Nevertheless, our approach differs from previous work in two aspects. First, the purpose is not to identify the impact of changing a source code entity, nor checking which of the entities that are likely to be affected is missing a modification, but to find the reasons behind the modifications to an application. Second, we propose a new evaluation framework to assess the value of reasons in terms of comprehension, and usefulness to locate related methods, rather than precision and recall of a prediction.

Robillard and Dagenais study is the closest to this study [5]. They analyzed to what extent the information contained in clusters of co-changing source code elements (fields or methods) was useful for change tasks. However, they cluster commit transactions that have common code elements while we cluster methods whose changes overlap in at least 80%. Moreover, their main assumption contradicts ours. They assume that by establishing transitive co-change relations it is possible to identify hidden logical dependencies, while we assume that there is no such transitiveness in co-change relations and that, in fact, the reasons behind a pair of co-changing methods can be different to the reason behind a second pair of co-changing methods even if the two pairs share one method. Finally, the information they recommended also could only support a minority of change tasks. In contrast, between 20% and 71% of our reasons are plausible depending on the configuration.

VII. Threats to Validity

This section discusses the main threats to the validity of our results.

A. Construct validity

Our approach is based on the hypothesis that changes ripple along dependencies between methods. While in theory this is the case, methods might change together for reasons other than their "logical" coupling, for example, a developer might bundle together changes to all methods under his control. Similarly, we mine logical changes from source code repository systems. It can be possible that the changes in the repository do not always correspond to logical changes, which would obscure the underlying reasons for the changes. We mitigate these threats by considering semantic properties (such as comments present in the code), in addition to structural properties of the co-changing methods. We also group commit transactions that occur within a standard 3 minute sliding window, which has been used before to approximate logical changes. Our approach however, remains sensitive to particular commit practices.

B. Internal validity

In our experiment we characterize methods by extracting a number of properties, both structural and semantic. The particular choice of which characteristics to extract was dictated by two rationales: First, structural properties evidence explicit dependencies that demonstrate the effect of coupling between entities (references to common types) and cohesion (methods in the same scope); while semantic properties evidence implicit dependencies (terms mentioned in Javadoc comments). Second, technical limitations: the parser we used does not resolve bindings, and provides no access to comments within a method. While performing the same experiment with a different set of properties, could lead different conclusions, we believe that the approach presented in this paper remains useful. Other choices that may have affected our study are: (1) the thresholds used for clustering recurrently co-changing methods, (2) not parsing identifiers into terms, and (3) ignoring the frequency of terms when extracting properties from javadocs. Further studies are required to evaluate the impact of these choices.

C. External validity

Regarding the generalization of our findings, the conclusions of this work are limited to the projects we have analyzed. Although we selected projects from different domains, they have roughly the same size and are implemented with the same programming language. The size of the applications

may have an effect on the relevance of the thresholds for defining clusters of recurrently co-changing methods. The programming language may also affect the importance of the structural relations analyzed. In any case the major issue with the generalization of our findings is the lack of more data points. Given that the analysis only takes into account two projects we do not know to what extent their commits profile is representative, and therefore we cannot generalize the results to other projects even if they are similar in terms of size and programming language.

D. Conclusion validity

In our study, we only draw conclusions referring to the best configuration for extracting reasons for co-change, which is based on the comparison of four metrics calculated for each of the data-sets that represent instances of different configuration options. The only configuration option that we did not vary was the definition of recurrently co-changing methods, which is in any case compared against reasons for methods that co-change once (i.e., commit-based reasons). Therefore, we are confident that the relationships found between configuration variables are justified.

VIII. Conclusions and Future work

In this paper we proposed an approach to automatically extract the rationale for a set of co-changing methods. We believe that the rationale for co-changes is orthogonal to the prediction of (co-)changes but is more likely to be remembered by developers because it helps to uncover design knowledge. The proposed approach is based on common properties shared by co-changing methods. We have shown that even though it is possible to extract reasons with good coverage, idiosyncrasy and uniqueness levels from single co-changes, the plausibility of the reasons is always improved when the set of methods analyzed co-changed recurrently. Our study also shows that reasons that take into account structural and semantic properties tend to outperform reasons that use only one type of property. This approach could be used not only to document the reasons behind co-change relations, but also, to predict the impact of adding a new code entity to the system (i.e., by comparing its properties against known co-change reasons). We plan to perform a more extensive experimentation on the thresholds used for identifying the clusters of recurrently co-changing methods, improving the quality of the semantic properties extracted, and assessing the usefulness of other types of properties (like transitive-calls, or belonging to the same slice).

Acknowledgment

Angela Lozano is financed by the CHaQ project of the Agentschap voor Innovatie door Wetenschap en Technologie. Carlos Noguera is funded by the AIRCO project of the Fonds Wetenschappeljik Onderzoek.

References

[1] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proc. Int'l Conf. on Software Maintenance (ICSM)*. IEEE Computer Society, 1998, pp. 190–198.

[2] A. Hassan and R. Holt, "Predicting change propagation in software systems," in *Proc. Int'l Conf. on Software Maintenance (ICSM)*. IEEE Computer Society, 2004, pp. 284–293.

[3] T. Girba, S. Ducasse, and M. Lanza, "Yesterday's weather: guiding early reverse engineering efforts by summarizing the evolution of changes," in *Proc. Int'l Conf. on Software Maintenance (ICSM)*. IEEE Computer Society, 2004, pp. 40–49.

[4] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 563–572.

[5] M. P. Robillard and B. Dagenais, "Recommending change clusters to support software investigation: An empirical study," *J. Softw. Maint. Evol.*, vol. 22, no. 3, pp. 143–164, Apr. 2010.

[6] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 351–360.

[7] A. Alali, H. Kagdi, and J. I. Maletic, "What's a typical commit? a characterization of open source software repositories," in *Proc. of the IEEE Int'l Conf. on Program Comprehension*, ser. ICPC '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 182–191.

[8] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," in *Proc. Int'l Conf. on Software Maintenance (ICSM'08)*. IEEE, October/Autumn 2008, pp. 227–236, 227-236.

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

[10] H. Gall, M. Jazayeri, and J. Krajewski, "Cvs release history data for detecting logical couplings," in *Proc. Int'l Workshop on Principles of Software Evolution (IWPSE)*, 2003, pp. 13–23.

[11] T. Zimmermann, S. Diehl, and A. Zeller, "How history justifies system architecture (or not)," in *Proc. Int'l Workshop on Principles of Software Evolution (IWPSE)*, 2003, pp. 73–83.

[12] T. B. V. Belle, "Modularity and the evolution of software evolvability," PhD thesis, The University of New Mexico, 2004.

[13] H. Kagdi, J. I. Maletic, and B. Sharif, "Mining software repositories for traceability links," in *Proceedings of the 15th IEEE International Conference on Program Comprehension*, ser. ICPC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 145–154.

[14] J. I. Maletic and A. Marcus, "Supporting program comprehension using semantic and structural information," in *Proc. of the Int'l Conf. on Software Engineering*, ser. ICSE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 103–112.

[15] M. P. Robillard, "Automatic generation of suggestions for program investigation," in *Proc. of the 10th European Software Engineering Conf. held jointly with 13th ACM SIGSOFT int'l symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 11–20.

[16] E. Hill, L. Pollock, and K. Vijay-Shanker, "Exploring the neighborhood with dora to expedite software maintenance," in *Proc. of the int'l conf. on Automated software engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 14–23.

[17] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. L. Collard, "Blending conceptual and evolutionary couplings to support change impact analysis in source code," in *Proceedings of the 2010 17th Working Conference on Reverse Engineering*, ser. WCRE '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 119–128.

[18] M. Gethers, H. Kagdi, B. Dit, and D. Poshyvanyk, "An adaptive approach to impact analysis from change requests to source code," in *Proc. of the Int'l Conf. on Automated Software Engineering*, ser. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 540–543.