Untangling Fine-Grained Code Changes

Martín Dias[1], Alberto Bacchelli[2], Georgios Gousios[3], Damien Cassou[1], Stéphane Ducasse[1]
1: RMoD Inria Lille–Nord Europe, University of Lille — CRIStAL, France
2: SORCERERS @ Software Engineering Research Group, Delft University of Technology, The Netherlands
3: Digital Security Group, Radboud Universiteit Nijmegen, The Netherlands

***Abstract*—After working for some time, developers commit their code changes to a version control system. When doing so, they often bundle unrelated changes (*e.g.,* bug fix and refactoring) in a single commit, thus creating a so-called tangled commit. Sharing tangled commits is problematic because it makes review, reversion, and integration of these commits harder and historical analyses of the project less reliable.**

Researchers have worked at untangling existing commits, *i.e.,* finding which part of a commit relates to which task. In this paper, we contribute to this line of work in two ways: (1) A publicly available dataset of untangled code changes, created with the help of two developers who accurately split their code changes into self contained tasks over a period of four months; (2) a novel approach, *EpiceaUntangler*, to help developers share untangled commits (aka. atomic commits) by using fine-grained code change information. *EpiceaUntangler* is based and tested on the publicly available dataset, and further evaluated by deploying it to 7 developers, who used it for 2 weeks. We recorded a median success rate of 91% and average one of 75%, in automatically creating clusters of untangled fine-grained code changes.

I. Introduction

Version Control Systems (VCS), such as Git and Subversion, allow programmers to control changes to source code and make it possible to find who made each software change, when, and where. This information is important to support both the coordination of developers working in teams [1] and the creation of many recommendation and prediction systems related to software quality [2].

Developers often bundle unrelated changes (*e.g.,* bug fix and refactoring) in a single commit [3], thus creating a so-called *tangled commit*, such as the following taken from Jaxen:[1]

```
------------------------------------------------------------
r1252 | elharo | 2006-11-09 [...] | 2 lines

Pulling getOperator up into BinaryExpr per Jaxen-169

[...]
Index: src/java/main/org/jaxen/expr/AdditiveExpr.java
===================================================================
--- src/[...]/AdditiveExpr.java        (revision 1251)
+++ src/[...]/AdditiveExpr.java        (revision 1252)
@@ -61,7 +61,7 @@
   *
- */public interface AdditiveExpr extends BinaryExpr
+ */
+public interface AdditiveExpr extends BinaryExpr
  {
-    String getOperator();
  }
```

The tangled commit above contains both a refactoring (the move of 'getOperator' to a different place, not shown in this extract), and code formatting (the move of an interface definition to its own line). Sharing tangled commits is problematic as they make code review, reversion, and integration harder and historical analyses of the project less reliable [4]. For example, even integrating the code formatting change included in the aforementioned commit (without the refactoring) would be a demanding task.

Untangling existing commits (*i.e.,* finding how to separate parts of a commit relating to different tasks) is an open research problem. Herzig and Zeller presented the earliest and most significant results in this area [3]: They implemented the first algorithm that can automatically untangle commits given artificially tangled ones.

In this paper, we expand on this previous work by: (1) working in an untyped setting where a part of the approach by Herzig and Zeller is inapplicable; (2) considering fine-grained code change information gathered during development (*e.g.,* time at which each line has changed and all versions of each line); and (3) evaluating the resulting approach both on data generated by programmers who manually label it and with programmers working on real-world development tasks.

The ultimate goal of our work is to help developers of dynamically-typed code share untangled commits. To that end, we: (1) asked 7 developers to manually cluster changes for each of their commits using a dedicated tool, for a period of 4 months; (2) manually validated the generated data, selecting the data recorded by two of these developers, and computed a number of features based on their fine-grained code changes; (3) modeled the problem of predicting whether two fine-grained changes belong together, with a variety of machine learning approaches, determined the most appropriate one, and identified the most significant features; (4) designed an algorithm that uses the machine learning result to propose an automatic clustering of any tangled commit and developed a corresponding tool, *EpiceaUntangler*; and (5) evaluated the effectiveness of our approach with developers who used *EpiceaUntangler* in their daily work for two weeks.

Our results show that three features are especially important to perform clustering of fine-grained code changes: (1) the time between two changes; (2) the number of other changes between two changes; and (3) whether the two changes modify the same class. By modeling these features with Random Forests [5], we identify whether two changes belong to the same commit with an accuracy of 95%, if training and testing on the same developer, and more than 88% if tested on a different developer. A set of 200 manually clustered fine-grained code changes (*i.e.,* the equivalent of a few days of work) was sufficient to reach good performance. When deploying *EpiceaUntangler* with new developers during their daily tasks, we recorded an average success rate of 75% and a median one of 91%.

[1]http://jaxen.codehaus.org, commit: svn-1252, 2006-11-09

978-1-4799-8469-5/15/$31.00 © 2015 IEEE

II. Problem Description

When developers want to share their work in a VCS, they will, more often than not, realize that they have done more than one activity, *e.g.,* fixed a bug, reformatted a method, and fixed a typo in a comment. Sharing everything in a single *tangled commit* is regarded as bad practice because it makes the following activities more difficult:

- *Review* – Reviewers have to understand the code changes of all the activities *at once* [6]–[8];
- *Reversion* – Developers have to revert all changes of a problematic commit even when only the code change of one activity is problematic [1];
- *Integration* – Integrators have to merge or reject whole commits, *e.g.,* they will typically reject a code formatting operation and a bug fix included in the same commit [9];
- *Historical analysis* – Researchers need to associate activities to files to conduct statistical analyses while, *e.g.,* mining software repositories [4].

A. Existing Solutions for Tangled Changes

To avoid tangled commits, developers could organize their work so that, at commit time, only one activity's code is to be shared. This requires frequent commits and interruptions in the developer's work flow [10]–[13]. Even with a lot of discipline, there will be times when a developer will have to split changed code into several commits.

To separate code from several activities into different commits, some tools (*e.g.,* 'git add') let the user selects which files and lines to commit first. Being line based, these tools share the following problems: (1) The code present at commit time might be *incomplete* [14]: Each change to a line shadows previous changes of the same line, thus making it impossible to commit the line as it was before the last change; (2) a commit resulting from a manual selection of a subset of all changed lines might be *invalid*: *e.g.,* a developer might commit the beginning of a function definition but not the end; and (3) changed lines are shown in the order they appear in their files irrespective of their modification time: This makes it difficult for developers to select lines changed closely in time.

A great source of inspiration for us comes from Herzig *et al.* [3], [4], who implemented an algorithm to automatically untangle commits. Their algorithm uses several *confidence voters* to decide whether two lines of a tangled commit should be put in the same cluster. They aggregate the results of each confidence voter into a single score, and then use the concepts of a multilevel graph-partitioning algorithm by Karypis and Kumar [15] to generate the clusters. Their voters include:

- 'FileDistance': the number of lines between the two lines if they are both in the same file;
- 'PackageDistance': the number of different package name segments within the package names of the changed files;
- 'CallGraph': the difference between the call graphs of the program with each line change applied separately;
- 'ChangeCouplings': the frequency with which the files both lines were changed into are committed together, using the work from Zimmermann *et al.* [2];
- 'DataDependency': a boolean indicating if the two lines read or write the same variable(s).

In the work by Herzig *et al.* we see the following limitations:

Dependence on static-analysis: The voters 'CallGraph' and 'DataDependency' rely on static analyses that might not be possible for dynamically-typed programming languages, or that might be available in a weaker form;

Incompleteness: The tangled commits used as input to the algorithm suffer from the incompleteness problem described earlier in this section: If a line is changed twice before a commit, the commit only contains the latest version of the line, shadowing a previous version of the line which could have been part of an untangled commit;

Artificiality: The validation by Herzig *et al.* relies on a classification of 7,000 existing commits done by the researchers without feedback from each project's experts. We believe that only the author of each commit can, at commit time, best organize his changes into untangled commits. Moreover, the untangling algorithm by Herzig *et al.* relies on the knowledge of the expected number of untangled commits for a particular tangled one. With the goal of helping developers creating untangled commits, we do not have access to this information.

B. Addressing the Current Limitations

In our work, we propose to alleviate the aforementioned limitations by (a) expanding the setting to a dynamically-typed environment where some kinds of analyses are not available; (b) using fine-grained code changes that we collect during development sessions; (c) relying on developer-approved data for the validation of untangling approaches. This results in the following requirements for the approach, *EpiceaUntangler*, that we present in this paper:

The Dynamically-Typed Setting: Whereas the approach of Herzig *et al.* relies on static analysis of Java programs to untangle commits, our approach helps developers to create untangled commits in an environment that is dynamically-typed. Certain types of static analysis, *e.g.,* accurate call graph analysis, is not possible for dynamically-typed languages. Therefore, our approach cannot rely on such static analysis.

Fine-Grained Changes: In modern integrated development environments (IDEs), tools can be notified each time a software artifact is changed and saved. As a result, a tool could listen to all fine-grained changes made by developers and, at commit time, present the developer a list of all the changes they have done. For example, a developer changing and saving the source code of a method 3 times will result in 3 fine-grained changes. This is in contrast with most tools that only present the latest version of each changed line; this requirement tackles the *incompleteness* limitation.

Developer-Approved Data: The untangling algorithm should be based on data created by developers who personally untangle the tangled commits that they produced in the first place. The final version of the approach should provide each developer, at commit time, with a list of the automatically untangled commits containing their fine-grained changes: Each developer could then reorganize these automatically-computed clusters of changes. Results must be validated by comparing the change clusters that are automatically computed against the reorganization done by the developer in a manual way.

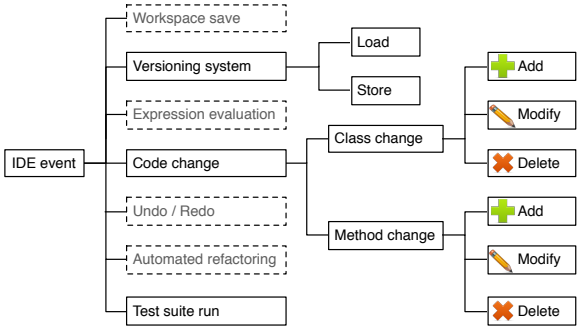


Fig. 1. IDE events recorded by Epicea; dashed ones are not considered.

III. Proposed Solution

In a nutshell, our solution is to develop an approach and associated tools to help developers share untangled commits. The tools log all the fine-grained changes made by developers as they change the source code. When a developer wants to commit her changes, the tool, based on an analysis of the recorded information, presents several automatically-computed clusters of changes: Each cluster represents a distinct activity of the developer since last commit. The developer may then add a comment to each cluster and, if necessary, adapt the automatic clustering (by adding/removing clusters and moving changes to different clusters). Once the developer validates the clusters, the tool generates one commit per cluster and publishes them to the repository. In the following section, we present our solution decomposed in individual parts.

A. Epicea: Event Modeler with Fine-Grained Changes

Central to our approach is the collection of fine-grained information. To conduct this task, we use Epicea [16], a tool we developed to model IDE events. In essence, Epicea listens to actions taking place in the IDE and records different types of events. A simplified version of the events recorded by Epicea is shown in Figure 1. Epicea records complete information of these events (*e.g.,* whether a test run failed), including a timestamp. As previous studies (both in Eclipse [17] and in Smalltalk [18]) showed that save-based recording produces reliable fine-grained code change data, we record code change operations (add, modify, and delete classes and methods) every time the user saves the code. Epicea is invisible to the user as there is no impact on performance. Epicea stores the collected data as a sequence of serialized objects in plain text files.

B. Voters

Once the data is collected, we have to characterize it in a way that it can be used for generating untangled changes. Similarly to Herzig *et al.* [3], [4], as a first step we model our clustering task as a binary classification problem: For all the potential pairs of recorded fine-grained changes, we want to determine whether they belong in the same cluster. To this end, we implement a number of features or, maintaining the term used by Herzig *et al.* [3], *voters*, which describe different relations between the considered changes. Our voters (detailed in Table I) span the following six dimensions:

1) *Code structure*: Although dynamic languages make it difficult to conduct static analysis, it is possible to compute basic relations. Our three voters in this dimension consider whether two changes happen in the same package, class, and/or method.
2) *Content*: This voter returns true if the two changes to a method are only source-code reformats, *i.e.,* if the *abstract syntax tree* of a method remains the same after a change on it. This voter should help linking changes regarding refactoring actions.
3) *Testing*: Epicea records test runs. The rationale of this voter is that two changes happening between runs of the same test could be related to the same task (*e.g.,* this should hold in the case of test-driven development).
4) *Spread*: These voters measure the distance between the two considered changes, considering time passed and number of other changes in between. We expect close changes to be more related.
5) *Message sending*: This dimension analyzes whether the changes involve related message sending (also known as 'method invocations', in languages such as Java or C#).
6) *Variable accessing*: This dimension computes relations between the variables accessed by the two changes: For example, a change that adds a new instance variable to a class may be related to a change that adds an usage of the same variable in a method.

The input of each voter is a pair of changes, and the output is of the type specified in column 'Type' of Table I.

C. Machine Learning Approaches

Our approach computes the values for each voter for each pair of changes (for performance reasons, we only consider fine-grained change pairs that are less than 3 days apart); to aggregate these values and train models that would predict whether two changes should be in the same cluster, we use machine learning (ML).

We consider three well-known machine learning algorithms that can handle binary classification [19]: (1) binary logistic regression ('binlogreg'), (2) naïve bayes ('naivebayes'), and (3) random forests [5] ('ranforest'). We chose these algorithms not only because they have been applied successfully to a number of data mining tasks related to software engineering, but also because they make quite different assumptions on the underlying data and model (*e.g.,* 'naivebayes' relies on the conditional independence assumption, *i.e.,* the value of a voter is unrelated to the value of the others, and 'binlogreg' requires each observation to be independent and linearity of independent variables and log odds), thus they can offer different interpretations. The choice of the most appropriate machine learning algorithm is based on the empirical data collected during the experiment.

This machine learning step takes as input the values computed by the voters for two particular changes, and it outputs the probability of the two changes belonging to the same cluster.

D. Clustering

The last necessary step in our approach is to take the output of the machine learning step, computed on each pair of changes, and aggregate it to form the clusters of changes for the user.

TABLE I. DIFFERENT VOTERS TESTED IN OUR INVESTIGATION.

Voter Name	Dimension	Type	Relation between the two considered changes
samePackage	Code structure	Boolean	They involve the same package.
sameClass	Code structure	Boolean	They involve the same class.
sameSelector	Code structure	Boolean	They involte a method with the same name (regardless its class).
bothCosmeticChanges	Content	Boolean	They are both cosmetic (*i.e.,* pretty-printing—both versions of the method return the same result).
sameTestRun	Testing	Boolean	They are modified between the same unit-test runs.
numberOfEntriesDistance	Spread	Numeric	How close they are in the history; the voter computes number of other changes between them.
timeDifference	Spread	Numeric	How close in time they are in the history; the voter computes the seconds between them.
reciprocalMessageSends	Message sending	Nominal	They invoke each other; it computes 0, 1 or 2 if, respectively, no, one, or both call the other.
numberOfSharedMessageSends	Message sending	Numeric	They share a number of the same message sends.
numberOfSharedMessageSendsInDelta	Message sending	Numeric	They add or remove a number of the same message sends.
numberOfVariableAccesses	Variable accessing	Numeric	One change modifies or adds definitions of instance variables, the other accesses some of them.
numberOfSharedVariableAccesses	Variable accessing	Numeric	They access a number of the same instance variable names.
numberOfSharedVariableAccessesInDelta	Variable accessing	Numeric	They start or stop accessing a number of the same variable names.

In this method, each change is initially considered to be a cluster of its own. Then pairs of clusters are successively selected by their maximum scores and merged. The result of this method is a *dendrogram*, which is a binary tree that represents the nested clustering of code changes. In this dendrogram, each non-leaf node has a *similarity level* that represents how similar are both children. In our problem, a similarity level of 1 corresponds to two clusters that must be merged, while a level of 0 corresponds to the opposite decision.

Finally, the desired clustering of code changes is obtained by cutting the dendrogram at some *similarity threshold*. Using a too low threshold produces too many small clusters, while a threshold that is too high produces a single cluster. The choice of the most appropriate similarity threshold depends on the change set and, similarly to the machine learning approach, is based on the empirical data collected during the experiment.

The output of this step is the set of independent clusters of fine-grained changes, which is eventually displayed to the user with a dedicated user interface.

IV. RESEARCH METHOD

In this section, we describe how we structure our research in terms of research questions, we present the research settings, and we outline our research method.

A. Research Questions

The ultimate goal of our work is to help developers of dynamically-typed code share untangled commits. For this we devise and test the approach we previously described to untangle code changes at a fine level of granularity. Accordingly, we structure our empirical investigation through the following three research questions:

RQ1: **Which voters are significant to untangle fine-grained code changes?**
With this question we aim to understand which are the most important voters in our untyped setting. To answer this research question, we consider the machine learning task of deciding whether two changes should belong to the same cluster. In doing so, we also determine which machine learning approach among the three we test, is better suited to model the problem through our voters.

RQ2: **How effective is a machine learning model based on the significant voters in untangling historical fine-grained code changes?**
Once we find the most significant voters and the best machine learning approach, we are interested to know their performance in predicting whether two changes should belong to the same cluster. We also want to investigate the effect asserted by individual developers' working styles on prediction performance; for this we train and test the machine learner on data generated by different developers (*e.g.,* training on one developer's data and testing on another developer's data).

RQ3: **How effective is a tool based on the best voters and machine learning approach, when deployed with developers working on their daily tasks?**
Finally, we want to devise an approach *EpiceaUntangler*, based on the best machine learner and voters, to generate clusters and present them with a graphical user interface. We want to test its effectiveness when deployed with participants (1) whose data should not have been used for training the classifier, and (2) who should be working on their usual development tasks.

B. Research Settings

Our study took place with professional developers, researchers, and students using the Pharo environment.[2] Pharo is an open-source dialect of Smalltalk and implementation of its programming environment. Pharo was forked from Squeak[3] in 2008 and it is rapidly evolving. Currently, Pharo has around 60 worldwide contributors, it is used by more than 15 universities to teach programming and by 10 research groups to build tools, and more than 50 companies are using it in production.

We chose Pharo as a case study for two main reasons: (1) the Pharo open-source community of developers has been receptive, since its inception, to welcome and thoroughly evaluate research tools (*e.g.,* [20]–[22]); and (2) the programming language, the development environment, and the versioning system are tightly integrated. The later feature allows for a fast prototyping of an approach to record fine-grained code changes and interaction with testing and the versioning system. The former feature allow us to collect fine-grained data about code changes and IDE interactions from participants doing real-world development

[2]Pharo: http://pharo.org/
[3]Squeak: http://www.squeak.org/

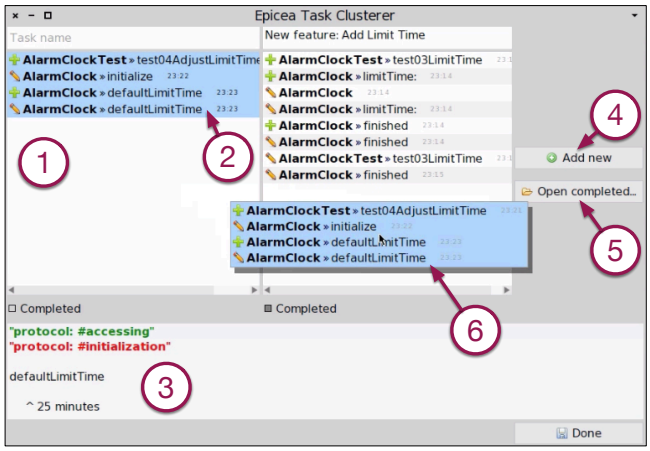


Fig. 2. UI used in training stage. The user manually clusters the changes.

work. It also enabled us to deploy our resulting tool with more participants to evaluate its results. Moreover, many research tools tested within Pharo later became integral part of the environment (*e.g.,* [23]); we want both to improve the state of the art in untangling code changes and to create an approach that can be used in real-world scenarios.

C. Research Steps

1) Fine-grained data generation and collection: To answer our first two research questions, we need a *ground truth* to train and test our voters and machine learning approaches. Such a ground truth should be a reliable dataset containing fine-grained code changes correctly split into tasks by their authors. To obtain this, we contacted 7 participants actively contributing to Pharo, including the first author of this paper. We asked them to install Epicea and to use the tool (*i.e., Epicea Task Clusterer* (ETC), Figure 2) that we devised to manually cluster their fine-grained code changes. We showed a screencast[4] demoing the tool to all the participants before they started using it, so that they could understand the goal of the experiment and adapt their workflow accordingly. Every time the participants decided to commit their code to the versioning system, during their normal work, the ETC's interface would appear (as in Figure 2) with a list of all the fine-grained changes, since the previous commit, that the user had to manually cluster into tasks.

TABLE II. PARTICIPANTS' INFORMATION

P_ID	current role	programming experience (in months) overall	industrial	with Pharo
Data generation and collection phase				
P1	Ph.D. student	168	60	48
P2	Ph.D. student	48	36	24
Evaluation in real-world development phase				
P3	Ph.D. student	180	18	36
P4	software engineer	132	72	13
P5	associate professor	72	12	24
P6	Ph.D. student	72	11	11
P7	software engineer	180	10	30
P8	software engineer	60	18	36

[4]Available at: https://www.youtube.com/watch?v=fQVWuMQUBew

In detail, the main user interface of *Epicea Task Clusterer* (shown in Figure 2) works as follows: In the top pane, each column (*e.g.,* Point 1) represents a task (to group an activity of the user), and each item in a column represents a code change (*e.g.,* Point 2). Each code change is in a *ClassName»methodName* format, and the icon shows the type of change (as in Figure 1). The bottom pane (Point 3) shows the details of the selected change, in a *unified-diff* format. The user can review the listed changes and perform three actions to specify the expected clustering for them: Add a new empty task/cluster (Point 4), reopen an already closed task/cluster (Point 5), and move changes between columns (with drag and drop, Point 6). Once the clustering task is completed, the user presses the button 'Done', and the interface disappears.

2) Data analysis and evaluation of voters: Once the participants concluded the data collection period of 4 months, we conducted exploratory data analysis [24] on the generated clustered changes. The data generated by five users was extremely sparse and inconsistent; these users confirmed this explaining that they could not afford the time required by *Epicea Task Clusterer* to review each change made during the experiment period. We removed this data and kept the data generated from the remaining two users (including the first author of this paper) whose features are described in the top half of Table II. Table III describes the resulting dataset (2devs).

TABLE III. DESCRIPTIVE STATISTICS OF DATASET 2DEVS

P_ID	Total number of changes	Total number of clusters	Changes per cluster Mean	Median	St. Dev.	Max
P1	15,175	298	50.9	8	153.1	1,582
P2	9,601	119	80.7	16	151.9	812

Using 2devs we answered RQ1 and RQ2. As previously detailed (Section III-C), we used machine learning to identify pairs of changes belonging to the same commit, by modeling it as a binary classification. For all potential pairs of changes in 2devs, we calculated values for all the voters in Table I and labeled with 'true' if the changes belonged to the same commit or 'false' otherwise. As our dataset was unbalanced (the false class overruled the true one by a ratio of 4:1), we adjusted to avoid overfitting. Models where thus trained with a ratio of 2:1 samples for the false and true class respectively.

Evaluation of voters. To evaluate each trained model, we used standard machine learning metrics [19], such as precision (prec), recall (rec), accuracy (acc), the Area Under the receiver operating characteristic Curve (auc) and the F-measure (f.measure). Models where trained with an increasing number of samples as input (10^4 to 10^6 samples) to determine the minimum number of samples required to obtain adequate performance. At each input size, we used random selection 10-fold cross validation to evaluate model stability and reported results based on the mean of the 10 runs. We selected the best classifier and applied a classifier-specific process to rank voters according to their importance in the classification. Then we incrementally trimmed the voter set starting from the least important feature until the performance of the classifier was severely impacted. Finally, we retrained the best classifier with the trimmed voter set and used that as our final prediction model. The final model was then exposed as a web service that *EpiceaUntangler* used to drive the change untangling process to answer RQ3.

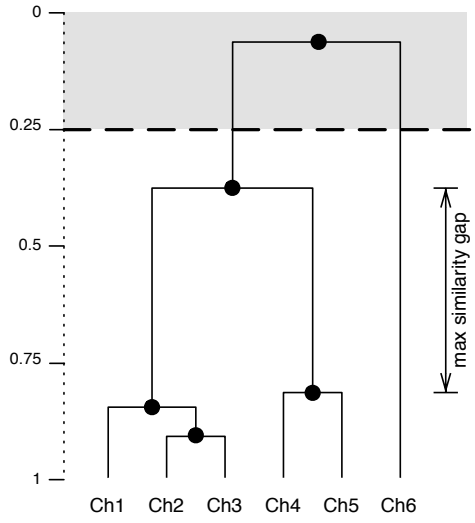


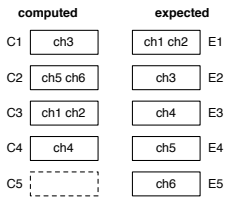
Fig. 3. Determining the *similarity threshold* to cut the dendrogram.

3) Deployment and evaluation with developers: Once we completed the creation and evaluation of the best ML approach and features on dataset 2devs, and obtained promising results, we created the corresponding implementation in *EpiceaUntangler*, a tool that developers can use in real-world development.

During developer's work, *EpiceaUntangler* records the fine-grained change information, exactly as done for the data collection phase. When the developer wants to commit, our approach computes the values for all the significant voters for each pair of code changes, and queries the web service implementing the final model of the ML classifier. For each pair, the web service returns a score between 0 and 1, indicating the probability that the two changes belong to the same cluster, according to the trained model. *EpiceaUntangler* aggregates all the scores to form clusters using agglomerative hierarchical clustering method (see Section III-D). This method outputs a dendrogram, which has to be cut at some similarity threshold to obtain the clusters of changes. We created a testbed with 'change set-expected clustering' pairs whose purpose is to help us to conceive a good function for obtaining the similarity threshold for cutting the dendrogram. In Figure 3 we illustrate the function. The similarity threshold we chose corresponds to the maximum similarity gap between all nodes whose similarity level is less than 0.25. The intuition behind taking the maximum similarity gap is that continue merging code changes together is not worth, because the meaningful clusters have already been detected. The reason to use 0.25 as a lower bound is that we observed from data that such a low likelihood indicates in most cases changes that should not be merged.

This process happens in the background: After the developer decides to commit, she sees an interface similar to that used to generate the data for 2devs (Figure 2), with the difference that the clusters are already pre-computed by the tool. Then, the user browses the clusters and reorganizes the changes in case the pre-computed clusters are wrong.

Evaluation of clustering. To conduct this evaluation, we recruited six participants, whose features are described in the bottom half of Table II. They all used *EpiceaUntangler* for 2 weeks. To evaluate the clustering, each participant was asked to confirm whether the automatic clustering was correct; if not they could rearrange changes to the correct clusters. We used the resulting data to evaluate the accuracy of our approach.



	E1	E2	E3	E4	E5
C1	0	1	0	0	0
C2	0	0	0	½	½
C3	1	0	0	0	0
C4	0	0	1	0	0
C5	0	0	0	0	0

Fig. 4. Comparison between a computed clustering and an expected clustering. On the left-hand side, each box represents a cluster of changes. The computed clustering contains 4 clusters labeled from $C1$ to $C4$ (cluster $C5$ is a *virtual cluster* to ease comparison). The expected clustering has 5 clusters: $E1$ to $E5$. On the right-hand side, the matrix shows the corresponding Jaccard indexes.

To measure the success rate of our approach, *i.e.,* how similar the *computed clustering* (from our algorithm) is to the *expected clustering* (from the developer), we need to know the ratio between the number of successfully clustered changes and the total number of changes. To know if a change has been successfully clustered, we must find which computed cluster best matches which expected cluster.

Figure 4 shows a sample comparison between a computed clustering and an expected clustering. The matrix on the right represents the *Jaccard indexes* computed for each pair of clusters; this index is defined as using the following formula:

$$J_{CiEj} = \frac{|Ci \cap Ej|}{|Ci \cup Ej|}$$

This Jaccard index represents how much two sets coincide. It ranges from 0 to 1, where 1 means the two sets are equal (*e.g.,* $C3$ and $E1$ in Figure 4) and 0 means the two sets have nothing in common (*e.g.,* $C4$ and $E2$ in Figure 4).

From the resulting matrix we want to know which computed cluster matches which expected cluster. This can be obtained by maximizing the sum of the Jaccard indexes over all permutations. For the sample in Figure 4 the maximum sum over all the permutations (3.5) is attained for this set of pairs:

$$Matching = \{(C1, E2)(C2, E4)(C3, E1)(C4, E3)(C5, E5)\}$$

We compute the success rate of our algorithm using the following formula:

$$SuccessRate = \frac{\#SuccessfullyClusteredChanges}{\#Changes}$$

A change ch_i is *successfully clustered* if the computed and expected clusters that contain ch_i are in the same pair of the $Matching$ set. In Figure 4, all changes are successfully clustered except $ch6$. This gives us a success rate of $5/6 = 0.83$.

V. Results

In this section we answer our research questions, by describing the results we obtained in our evaluations.

A. What Are the Dominant and Significant Voters?

As a first step to answer our first research question, we use all the machine learning approaches we consider on the collected data and we evaluate whether an approach performs undoubtedly better. Table IV reports the results of the classification performance of each machine learning approach for predicting whether two changes belong together, using a training size of $n = 320{,}000$ pairs (or 800 fine-grained changes), on the 2devs dataset. Overall, and across all metrics, the Random Forests algorithm delivers the best results, by a large margin. The high REC measurement of the 'binlogreg' result can be justified by its equally low PREC; the classifier marks most of the file changes as belonging in the same cluster, but few of those decisions are correct.

TABLE IV. Classification performance on 2devs by approach

Classifier	AUC	ACC	PREC	REC	F.MEASURE	G.MEAN
'binlogreg'	0.92	0.68	0.43	**0.96**	0.60	0.76
'naivebayes'	0.88	0.65	0.41	0.94	0.57	0.73
'ranforest'	**0.99**	**0.96**	**0.96**	0.88	**0.92**	**0.93**
'ranforest-trimmed'	0.98	0.95	0.96	0.82	0.88	0.90

Once we established that 'ranforest' delivers the best results, we assessed the importance of each voter for its classification result. We used the process suggested by Genuer *et al.* [25]. Specifically, we run the algorithm 50 times on a randomly selected sample of 10^6 change pairs, using a large number of generated trees (500) and trying 5 random variables per split. Then, we used the mean across 50 runs of the Mean Decrease in Accuracy metric, as reported by the R implementation of the *ranforest* algorithm, to evaluate the importance of each feature. The results can be seen in Figure 5. The three most important voters are: (1) the time difference between the changes, (2) the ordered distance of the changes, (3) and whether the changed code belonged in the same class. We cannot make inferences about whether the effect of each voter is positive or negative to the response class; nevertheless, we believe that the results are indicative of the task-based nature of software development.

B. How Effective Is Random Forests with the Dominant Voters?

We answer our second research question by using only the three most important voters to train the prediction model. The prediction results are reported in Table IV, marked as 'ranforest-trimmed'. We see that even with just those voters we obtain very good prediction results: The new model is within 3% of the performance of the model trained in all metrics.

Furthermore, we analyze the impact of the developer who made the changes on training and testing. We expect that behaviors of developers might be different and have a significant impact on the model.

We start showing that we obtain the best results when we train and test from data generated by the same developer (the 'intradev' dataset in Figure 6). This confirms our hypothesis that the behavior of the specific developer has an impact on the model and the results. Furthermore, we see that the results are not equally good when training with data from one developer and testing on the other (the 'crossdev' dataset); moreover we see that as we increase the training size, there is a drop in performance. This can be attributed to overfitting the model to the working habits of each individual developer. Finally, we see that we can train accurate models by combining data from multiple developers. In the 'combined' dataset, we combine

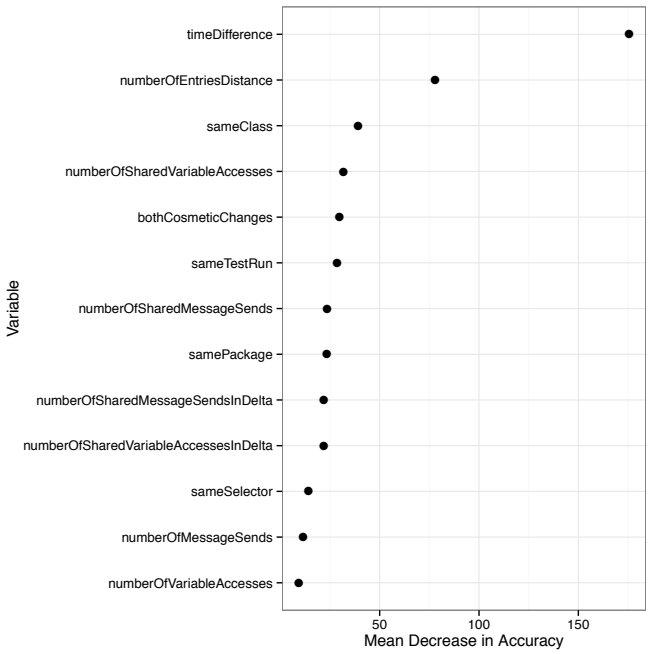


Fig. 5. Voter importance for the random forest classifier.

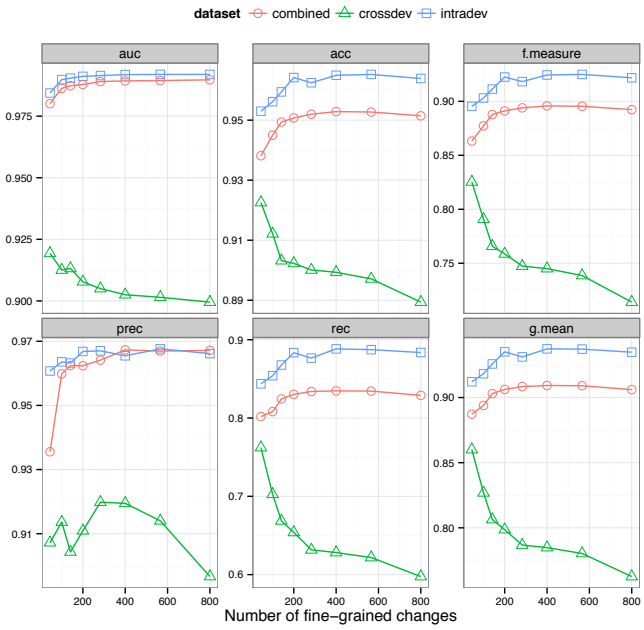


Fig. 6. Dataset performance metrics

the data generated by both developers and use this to train the model; this means that training and testing data is taken from both samples. Figure 6 shows that this dataset reaches high and stable results; and overfitting seems not present.

What is interesting to note is that the number of fine-grained changes required for training in both the *combined* and *intradev* cases is low: with 200 changes we can obtain prediction results only 2% worse (in terms of acc) on average than if we train with 800 changes. As 200 fine-grained changes are the equivalent of a few days of work,[5] we have encouraging evidence that an accurate model can be trained fast and deliver good results for a single developer. Moreover, a pre-trained model with data from multiple developers might be enough as a starting point for an untangling tool, which could then be trained to a particular developer's working habits.

Overall, the results show that using the random forest algorithm, a randomized set of about 200 fine grained changes and a few easy-to-calculate voters, we can train a prediction model that can identify whether two changes belong in the same commit with an accuracy of 95% for a single developer.

C. How Effective Is EpiceaUntangler for Developers?

We answer research question three by deploying *EpiceaUntangler* with developers and recording whether the clustering that it proposes corresponds to participants' expectations. The dataset devEval, resulting from this evaluation is described in Table V. We notice that not all the developers coded full time during the two weeks, thus some produced fewer changes.

TABLE V. DESCRIPTIVE STATISTICS OF DATASET DEVEVAL

P_ID	Total number of		Changes per cluster			
	changes	clusters	Mean	Median	St. Dev.	Max
P3	350	22	15.9	11	13.5	42
P4	826	28	29.5	3.5	50.9	228
P5	200	13	15.4	10	17.3	65
P6	166	12	13.8	6.5	15.6	47
P7	347	18	19.3	7	27.8	88
P8	162	11	14.7	10	12.7	37

We compared each cluster we proposed to the cluster that the participant eventually judged as correct to be committed. The histogram in Figure 7 shows the frequency of the obtained results: We observed a median[6] success rate of 0.915 and an average of 0.753 with a standard deviation of 0.30. By inspecting the instances with a success rate in the range [0,0.4] we could not pinpoint any systematic error; we plan to further address these cases in future work.

We asked developers their opinion on the tool and received diverse feedback. Most developers were positive [P3, P4, P6, P8], *e.g.,* P3 expressed the feeling that "*EpiceaUntangler guesses correctly the clusters of changes, also in a big commit were I had 10 different clusters,*" and P4 said: "*It works good in many cases, especially for not so big change sets.*" At the same time, most developers [P4, P5, P6, P8] expressed concerns with the large amount of fine-grained information to be processed; they explained that it adds too much noise to see not only the last state of a method but also all the intermediate modifications

[5]From the data we recorded, 200 fine-grained code changes correspond to two to five days of work, depending on the developer's style and pace.

[6]The results are not normally distributed, thus we report the median value.

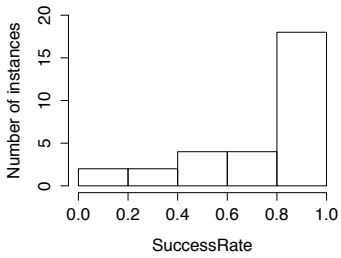


Fig. 7. Frequency of success rate of *EpiceaUntangler* clustering approach

to it, especially when belonging to the same cluster. In the words of P8: "*It was a bit painful to check everything.*"

Some participants suggested improvements to the user interface: For example, P7 said that he "*would like to option to delete tasks in the UI*", and P6 said: "*I would like to type a name for each task in the UI, as a reminder while I cluster.*"

VI. DISCUSSION

In this section we discuss our results and show how we mitigate the threats that endanger them.

A. Results

In the first research question, we asked which voters, or features are significant to untangle fine-grained code changes. Despite the fact that we implemented voters along six dimensions, only two dimensions were significant and contributed to most of the outcome: *code structure* and *spread*. In particular, the latter dimension has the greatest impact, by a large margin; the only significant voter in the former dimension measured whether the two changes were happening in the same class. This implies that these voters can be applied to other object-oriented programming languages regardless of whether they use types or not. This is a ripe opportunity for testing the approach with different languages and in different settings. Moreover, although we have no information about the significance of the voters implemented by Herzig *et al.* [3], [4], studies can be designed and carried out to determine if and how untangling effectiveness increases as a result of combining their voters with our significant ones.

We were surprised by the low impact of many of the voters in the untangling task: We expected message sending and variable access, as well as testing information, to contribute more. Since our initial data analysis was conducted with changes collected by only two developers, a further study with a larger set of developers for generating training data would be useful to confirm or alter this result.

In the second research question, we asked how effective is the best performing machine learning algorithm (*i.e.,* random forests) when used with the most significant voters. The results were overall very good. Expectedly, we achieved the best results when training and testing on data from the same developer, nevertheless cross-developers results are promising and merged-developers results do not show overfitting; in addition, approximately 200 fine-grained code changes were enough to reach most of the effectiveness. This implies that training on more developers is necessary to achieve a more

general approach, and there seem to be no risk of overfitting by doing it. Moreover, ideally every user should train the approach on her own programming behavior; this seems reasonable since the training is effective with as little as a few days of work.

In the third research question, we investigate the effectiveness of the whole approach when deployed to programmers. Considering that the recruited participants were not used for the training phase, the results are in line with the effectiveness measured for RQ2. One of the most recurring complaints was about the large number of changes to be verified and sorted at every commit. This is due to the fact that we showed all the fine-grained changes recorded, thus also intermediate states for the same method (when the developer saved multiple times). We expect this information overload problem to be mitigated once the approach is stable enough to work correctly in most cases. Nevertheless, we see a good opportunity for further investigating how certain fine-grained code changes can be omitted, without losing relevant information that would lead to the incompleteness discussed in Section II-A. Moreover, valuable comments were provided about the UI of *EpiceaUntangler*. The UI evaluation goes beyond the scope of this paper, but improving the UI can help to have an impact on reducing the information overload of fine-grained code changes.

B. Threats to Validity

Internal Validity. Our models and feature selection process are based on a dataset generated through the actions of two developers. While we have combined the actions of the developers and shown that they provide very good prediction performance and the evaluation of the *EpiceaUntangler* has been overwhelmingly positive, it is possible that our findings are biased towards the two developers' working habits.

Bias with respect to developer working habits might also occur in our selection of evaluation subjects. To reduce this risk, we selected diverse developers, all of them working in different projects and even in different physical locations. Thus, we believe the participants represent a heterogeneous enough population of Pharo developers.

Construct Validity. The notion of task is ambiguous. In particular, each participant can interpret the task granularity differently. For example, consider a single bug fix which is intended to fix two broken features. The participant could consider the changes either as two individual tasks, or everything as a single bug-fixing task. For mitigating this risk, we prepared a screencast with an example for users trying to establish a common criterion for task granularity. Moreover, we kept in close contact with users for answering any doubt. However, this ambiguity in the definition of task does not reduce the precision of our success metric for answering RQ3 (*SuccessRate*), since it represents each user expectation: it compares *EpiceaUntangler*'s clustering with the participant's expected clustering.

The clustering computed by *EpiceaUntangler* may have influenced participants. When users had to evaluate the computed clustering (as shown in Figure 2), the initial clustering might have biased their answers.

External Validity. We used a specific platform (Pharo) and language environment (Smalltalk) to facilitate our study. A specific language may dictate a specific working style. For example, in a typed language setting, an IDE would immediately mark as erroneous cases where a type signature has changed and not all uses have been adapted, therefore prompting the developer to fix such cases. Therefore our results may not be generalizable to all languages or working environments.

VII. Related Works

The impact of tangled changes has been reported in several contexts: The inspiring work by Herzig *et al.* [4], reported that at least 16.5% of all source files in the datasets they considered were incorrectly associated with bug reports when ignoring the existence of tangled change sets. In a large-scale study done at Microsoft on how developers understand code changes, Tao *et al.* reported that developers find it important for understanding to decompose changes into the individual development issues, but there is currently no tool support for doing so [6]. Bacchelli and Bird reported that tangled changes in code to be reviewed often cause low quality reviews or require longer time to review [7].

Herzig *et al.* were the first to implement an algorithm to automatically generate untangled commits given a tangled one. Their work greatly inspired our research. However, we see some limitations to their work that we explained in Section II: static-analysis dependency, incompleteness, and artificiality. The main differences with our work is that: (1) we count with fine-grained timing information of code changes as well as IDE events like test runs; (2) we work in a dynamically-typed language; (3) we evaluated our approach with developers.

Another source of inspiration comes from Robbes, who created a fine-grained change model of software evolution based on three principles [26]: (1) a program state needs to be represented accurately by an Abstract Syntax Tree (AST); (2) a program's history is a sequence of changes, each one producing a program state (an AST) and changes can be composed into higher-level changes; (3) changes should be recorded by the IDE as they happen, not recovered from a VCS. Robbes *et al.* show how a fine-grained change model can better detect *logical coupling* between classes [27]. Their article presents new measures of logical coupling that we consider as a future extension of our voters.

Steinert *et al.* propose CoExist, an approach and associated tool set to navigate the different states of a project based on its fine-grained changes [13]. CoExist's tool suite allows for reverting any fined-grained change at the project level, comparing different states of a program, localizing the cause of a failing test in the development history, and reassembling changes to share untangled commits. Automatic clustering of dependent fine-grained changes to create untangled commits is left as future work. Our work can be seen as an extension of CoExist tool suite in this direction, despite its totally unrelated implementation.

Wloka *et al.* presented a program analysis technique to identify committable changes that can be released early, without causing failures of existing tests [28]. Wloka remarks that an untangling algorithm would clearly benefit from having a model with a more accurate concept of change to add context information for individual change operations. Beyond our 'Same Test Run' voter, we leverage more the results of unit-test execution to cluster related changes.

VIII. Conclusion

In this paper, we have devised and evaluated *EpiceaUntangler*, an approach whose ultimate goal is to help developers share self-contained changes that are well-decomposed into individual tasks. We build on the shoulders of others, and expand previous work by: (1) Working in an untyped language setting where static code analyses are more limited; (2) considering fine-grained code change information gathered during development; and (3) evaluating the resulting approach both on data generated by programmers who manually labeled it and with programmers working on real development tasks.

Our results show that three features are especially important to perform clustering of fine-grained code changes: the time between the changes, the number of other modifications between the changes, and whether the changes modify the same class. By testing the features on historical data manually labeled by developers, we obtained good results (over 88% of accuracy in the worst case) in determining whether two changes should be together. When deploying our approach with new developers, we obtained a median success rate of 91%.

Overall, this paper makes the following main contributions:

1) An analysis of the current points for improvement in the state of the art in untangling code changes.
2) A publicly available[7] dataset of fine-grained code changes collected by recording the development sessions of two developers over the course of four months, and the corresponding manual clustering.
3) The creation of different features/voters and their evaluation, based on the aforementioned dataset, using machine learning approaches to model and classify pairs of fine-grained code changes, resulting in good accuracy results.
4) The creation of an approach, *EpiceaUntangler*, and corresponding tool implementation, *Epicea Task Clusterer*,[8] to untangle fine-grained code changes into clusters based on the three best voters and the best performing machine learning algorithm.
5) The deployment and a two-week evaluation of *EpiceaUntangler* with developers with good results.

Acknowledgements

We thank our study participants for their feedback and the European Smalltalk User Group[9] for its support.

References

[1] A. Guzzi, A. Bacchelli, Y. Riche, and A. van Deursen, "Supporting developers' coordination in the ide," in *In Proceedings of CSCW 2015 (8th ACM Conference on Computer Supported Cooperative Work and Social Computing)*. ACM, 2015, p. in press.

[2] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, Jun. 2005.

[3] K. Herzig and A. Zeller, "Untangling changes," *Unpublished manuscript*, Sep. 2011. [Online]. Available: https://www.st.cs.uni-saarland.de/publications/files/herzig-tmp-2011.pdf

[4] ——, "The impact of tangled code changes," in *Proceedings of 10th Conference on Mining Software Repositories*. IEEE, 2013, pp. 121–130.

[5] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[6] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes?: An exploratory study in industry," in *Proceedings of FSE 2012 (20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering)*. ACM, 2012.

[7] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of ICSE 2013 (35th ACM/IEEE International Conference on Software Engineering)*, 2013, pp. 712–721.

[8] G. Gousios, A. Zaidman, M.-A. Storey, and A. Van Deursen, "Work practices and challenges in pull-based development: the integrator's perspective," in *Proceedings of ICSE 2015 (37th International Conference on Software Engineering)*, 2015, p. in press.

[9] V. Uquillas Gómez, "Supporting integration activities in object-oriented applications," Ph.D. dissertation, Vrije Universiteit Brussel - Belgium & Université Lille 1 - France, Oct. 2012.

[10] "Subversion best practices," Apache, Software Foundation, 2009.

[11] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.

[12] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[13] B. Steinert, D. Cassou, and R. Hirschfeld, "CoExist: Overcoming aversion to change," in *DLS'12: Proceedings of the 8th Dynamic Languages Symposium*. ACM, 2012, pp. 107–118.

[14] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig, "Is it dangerous to use version control histories to study source code evolution?" in *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP)*, 2012.

[15] G. Karypis and V. Kumar, "Analysis of multilevel graph partitioning," in *Proceedings of Supercomputing 1995 (ACM/IEEE Conference on Supercomputing)*. ACM, 1995.

[16] M. Dias, D. Cassou, and S. Ducasse, "Representing code history with development environment events," in *IWST'13: International Workshop on Smalltalk Technologies 2013*, 2013.

[17] L. Hattori and M. Lanza, "Syde: A tool for collaborative software development," in *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, 2010, pp. 235–238.

[18] R. Robbes and M. Lanza, "A change-based approach to software evolution," *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 166, pp. 93–109, Jan. 2007.

[19] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*. Springer, 2001.

[20] L. Renggli, S. Ducasse, T. Gîrba, and O. Nierstrasz, "Domain-specific program checking," in *Proceedings of the 48th International Conference on Objects, Models, Components and Patterns (TOOLS'10)*, ser. LNCS, J. Vitek, Ed., vol. 6141. Springer-Verlag, 2010, pp. 213–232.

[21] A. Hora, N. Anquetil, S. Ducasse, and M. T. Valente, "Mining system specific rules from change patterns," in *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE'13)*, 2013.

[22] A. Hora, A. Etien, N. Anquetil, S. Ducasse, and M. T. Valente, "Apievolutionminer: Keeping api evolution under control," in *Proceedings of the Software Evolution Week (CSMR-WCRE'14)*, 2014.

[23] T. Verwaest, C. Bruni, M. Lungu, and O. Nierstrasz, "Flexible object layouts," in *Proceedings of OOPSLA '11 (26th International Conference on Object-Oriented Programming, Systems, Languages, and Applications)*. ACM, 2011, pp. 959–972.

[24] C. O'Neil and R. Schutt, *Doing Data Science*. O'Reilly, 2013.

[25] R. Genuer, J.-M. Poggi, and C. Tuleau-Malot, "Variable selection using random forests," *Pattern Recognition Letters*, vol. 31, no. 14, 2010.

[26] R. Robbes, "Of change and software," Ph.D. dissertation, University of Lugano, Switzerland, 2008.

[27] R. Robbes, D. Pollet, and M. Lanza, "Logical coupling based on fine-grained change information," in *Reverse Engineering, 2008. WCRE'08. 15th Working Conference on*. IEEE, 2008, pp. 42–46.

[28] J. Wloka, B. Ryder, F. Tip, and X. Ren, "Safe-commit analysis to facilitate team software development," in *Proceeding ICSE 2009 (31st International Conference on Software Engineering)*, 2009, pp. 507–517.

[7] Available at: http://dx.doi.org/10.6084/m9.figshare.1241571

[8] Available at: http://smalltalkhub.com/#!/~MartinDias/EpiceaTaskClusterer

[9] ESUG: http://esug.org