Replaying IDE Interactions to Evaluate and Improve Change Prediction Approaches

Romain Robbes
University of Chile

Damien Pollet
Inria – Université de Lille 1, France

Michele Lanza
University of Lugano, Switzerland

***Abstract*—Change prediction helps developers by recommending program entities that will have to be changed alongside the entities currently being changed. To evaluate their accuracy, current change prediction approaches use data from versioning systems such as CVS or SVN. These data sources provide a coarse-grained view of the development history that flattens the sequence of changes in a single commit. They are thus not a valid basis for evaluation in the case of development-style prediction, where the order of the predictions has to match the order of the changes a developer makes.**

We propose a benchmark for the evaluation of change prediction approaches based on fine-grained change data recorded from IDE usage. Moreover, the change prediction approaches themselves can use the more accurate data to fine-tune their prediction. We present an evaluation procedure and use it on several change prediction approaches, both novel and from the literature, and report on the results.

I. Introduction

Integrated Development Environments (IDEs) such as Eclipse or VisualStudio have had a major impact on how developers see and modify code. There is a paradigm shift taking place, with people departing from the notion that development is equivalent to writing source code text [1], towards a more "agile" view where systems are composed and continuously restructured and refactored. This is done using a plethora of tools provided by the IDE themselves, but also by third-party plugins. In the context of modern IDEs, recommender systems are gaining importance and gradually fulfilling Zeller's wish for "assistance" expressed during his MSR 2007 keynote talk, where he stated that developers in the future would be aided by many small visible and invisible IDE assistants, also known as recommenders.

Among the many recommenders that have already been built, change predictors play a prominent role. They assist developers and maintainers by recommending entities that may need to be modified alongside the entities currently being changed. Change prediction is useful for both software maintenance and forward engineering.

For *software maintenance*, change predictors recommend changes to entities that may not be obvious [2], [3]. In a software system, there often exist implicit or indirect relationships between entities [4]. If one of the entities in the relationship is changed, but not the other, subtle bugs may appear that are hard to track down.

For *forward engineering*, change predictors serve as productivity enhancing tools by easing the navigation to the entities that are going to be changed next. In this scenario, a change predictor maintains and proposes a set of entities of interest to help developers focus the programming tasks [5], [6], [7].

There is a need for a benchmark with which approaches can be compared. So far, maintenance-mode change prediction has been validated using the history archives of software systems as an oracle. For each transaction, the predictor proposes entities to change based on parts of the transaction. The suggestions are then compared with the remainder of the transaction. This can be considered a benchmark as the process is objective, measurable, cheap to run and precise. In forward engineering, where the system is actively developed and changes at a quick pace, the approach is not satisfactory. Intuitively, it is hard to split the changes in a commit in two when there is a large number of them: A finer grained level of detail is needed. As a consequence tools adapted to the forward engineering use case are assessed through comparative studies involving developers which are much harder to repeat and rerun, or recorded IDE interactions that are too shallow to evaluate a large array of approaches.

We present a benchmark for forward engineering, development-style change prediction. The benchmark consists of a number of fine-grained development histories, where we recorded *each* elementary change made to the systems while they were developed. Our detailed histories are thus unaffected by large and unordered transactions. In a nutshell, our benchmark allows us to reproducibly *replay* entire development histories, thus providing (close to) real life data without needing to perform comparative studies. Moreover, we introduce a procedure with which change prediction approaches can be evaluated using our benchmark data, and provide a comparative evaluation of several change prediction approaches. The contributions of this paper are:

- The definition of a benchmarking procedure for development-style change prediction approaches based on fine-grained, recorded IDE interactions, instead of coarse-grained, file-based SCM transactions. We replay the development of the systems in the benchmark in a very fine-grained manner. This emulates the development with a much greater accuracy than previous benchmarks based on SCM histories.
- The definition of a performance measurement adapted to development-style change prediction, based on the information retrieval measure of *cumulative gain*.

- The replication of several change prediction approaches in the literature, evaluated on the same dataset.
- The introduction of novel approaches and variants, making use of the additional fine-grained data we recorded to improve the performance of change prediction approaches beyond the state of the art.

Structure of the paper. Section II describes various change prediction approaches existing in the literature in the two change prediction styles. Section III justifies and presents our benchmark for change prediction approaches, based on fine-grained, recorded change histories. Section IV details the approaches we evaluated with our benchmark and presents the evaluation results, which we discuss in Section V, before concluding in Section VI.

II. Related Work

We survey maintenance-oriented and development-oriented change prediction approaches, and discuss strengths and shortcomings in their evaluation procedures.

A. Maintenance-oriented Approaches

There are two main trends of maintenance-oriented approaches: Historical approaches and approaches using the structure of the system through coupling metrics.

Historical Approaches. Zimmerman et al. [3] mined the CVS history of several open-source systems to predict software changes using the heuristic that entities that changed together in the past are going to change together in the future. They reported that on some systems there is a 64% probability that among the three suggestions given by the tool when an entity is changed, one is a location that indeed needs to be changed. Their approach works best with stable systems, where few new features are added. Changes were predicted at the class level, but also at the function (or method) level, with better results at the class level.

Ying et al. employed a similar approach and mined the history of several open source projects [2]. They classified their recommendations by interestingness: A recommendation is obvious if two entities referencing each other are recommended, or surprising if there was no relationships between the changed entity and the recommended one. The analysis was performed at the class level. Sayyad-Shirabad et al. also mined the change history of a software system in the same fashion [8], but stayed at the file level.

Change Prediction with Impact Analysis. Impact analysis has been performed using a variety of techniques; we only comment on a few. Briand et al. evaluated the effectiveness of coupling measurements to predict ripple effect changes on a 90 classes system [9]. The results were verified by using the change data in the SCM system over 3 years. One limitation is that the coupling measures were computed only on the first version of the system, as the authors took the assumption that it would not change enough to warrant recomputing the coupling measures for each version. The system was in maintenance mode. Wilkie and Kitchenham [10] performed a similar study on another system, validating change predictions over 130 SCM transactions concerning a system of 114 classes. 44 transactions featured ripple changes. Both analyses considered coupling among classes. Hassan and Holt compared several change prediction approaches over the history of several large open-source projects, and found that historical approaches have a higher precision and recall than other approaches [11]. Kagdi proposed a hybrid approach merging impact analysis techniques with historical techniques [12], but no results have been published yet.

Evaluating Maintenance-oriented approaches. Evaluations methods vary, but Hassan and Holt's Development Replay approach is a good representative of the techniques in general. Evaluation is based on the data found in SCM archives and proceeds as follows:

All the transactions in the SCM systems (e.g., the sets of files changed in each development sessions) are extracted from the repository and processed one by one. For each transaction, the set of changed entities C (files, classes or methods depending on the granularity of the approach) is split in two sets. One is the set of initially changed entities I and the other the set of entities to predict or oracle O.

To evaluate the quality of its predictions, the predictor is given the set I and returns a set of predictions P. This set is compared with the actual changed entities in O using the information retrieval measures of precision and recall. In this case, precision is the proportion of accurately predicted items in the predicted set ($\frac{P \cap O}{P}$) and recall is the proportion of predicted items out of the entities to predict ($\frac{P \cap O}{O}$). The predictor then has the opportunity to update its representation of the system when the actual changes are given to it. Depending on the approach, the predictor can use historical information, structural information, or both.

This evaluation approach has the advantage that a massive amount of data is available to test the approaches, in the form of the SCM archives of large open-source systems. Hassan and Holt carried out their experiments on 40 years of data. Such an amount of data gives great confidence in the results.

However, the SCM data used is not of optimal quality. This introduces imprecisions in the evaluation process. We documented the shortcomings of using SCM system for software evolution research [13]. SCM systems such as CVS and Subversion are file-based (they version files to be compatible with any kind of documents) and snapshot-based (they version snapshots of the system, and do not monitor the developers as they work).

Being file-based means that the unit of change for the SCM system is the file. To evaluate a change predictor working at the level of classes, or methods, extra processing is necessary. This is possible in theory but not often done in practice, even if the results would be more relevant to a developer. Of the approaches we surveyed, only Zimmer-

mann's has the ability to predict changes at the method level.

Being snapshot-based means that fine-grained information about the exact content and sequence of the changes is missing. It is *impossible* to recover from an SCM system this kind of information as it was never in the versioning system to begin with. Most versioning systems simply take snapshots of the code base at the developer's demand. Thus SCM-based evaluation of change prediction assumes that the order of the changes is irrelevant, and assigns entities to the initial set I and the oracle set O arbitrarily. Ignoring the sequence of changes makes the change prediction farther from a real-life scenario that it needs to be.

Further, valuable information about the changes is withholded from the change predictors. These can not exploit fine-grained changes and structural information to a greater extent, which may hamper the accuracy of their predictions.

B. Development-oriented approaches

The goal of development-oriented change prediction approaches is to ease the navigation to entities which are thought to be used next by the programmer. These approaches are based on IDE monitoring and predict changes from development session information rather than from transactions in an SCM system. They can thus better predict changes while new features are being built.

IDE-based approaches. Mylyn [6] maintains a task context consisting of entities recently modified or viewed for each task the programmer defined. It limits the number of entities the IDE displays to the most relevant, easing the navigation and modification of these entities. Mylyn uses a Degree Of Interest (DOI) model to select entities that the developers interacts with.

Navtracks [5] and Teamtracks [14] both record navigation events to ease navigation of future users, and are geared towards maintainance activities. Teamtracks also features a DOI model. Navtracks' recommendations are at the file level.

Heatmaps [7] highlights entities of interest in the IDE's code browser, based on previous activity in the environment. An entity's interest value is encoded in a color, ranging from blue (low value), to red (high value). Several heatmaps, emphasizing different aspects, are available.

Evaluating IDE-based approaches. Development-oriented prediction approaches are validated either with user studies, in which two groups of people perform the same task with or without the tool, or with recorded IDE interactions.

Mylyn has been validated by assessing the impact of its usage on the edit ratio of developers, i.e., the proportion of edit events with respect to navigation events in the IDE. Mylyn users spent more time editing code, and less time looking for places to edit. Teamtracks was validated with user studies, while Navtracks was validated both with a user study and with recorded navigation traces (around 35% of the recommendations were correct). Heatmaps were validated on recorded navigation traces as well.

While user studies yield a good degree of confidence in the results, they are not adapted to recommending algorihms that involve a fair amount of fine-tuning. User studies are expensive to perform, and are hard to reproduce. Lung et al. documented how they replicated an experiment involving human subjects, and found the process long and error-prone [15]. This makes it much harder to compare approaches to one another.

Recorded IDE interactions is hence an evaluation mechanism suited to the iterative improvement of a recommendation algorithm. However, most IDE-based monitoring tools withhold valuable information to the change predictors. The recorded IDE data is shallow: IDE-based approaches do not record actual changes, only change locations. This introduces some imprecision as all the change information is not available. Approaches mentioned above do not have a fully reified model of changes, i.e., these tools know *where* in the system something has changed, and *what* a developer is currently looking at, but they do not model *how* a piece of the system is being modified. Using the full structure of the program and the changes is hence not possible. Designing a repeatable evaluation method like Hassan and Holt's development replay is as comprehensive, as too much of the data is missing.

III. A Benchmark for Change Prediction

We claim that evaluation based on SCM data is too coarse and does not correspond to a real-life scenario. On the other hand, user-based evalution of development approaches are costly to setup and hard to compare to one another.

In contrast, we propose an evaluation approach which takes the best of the two worlds: we record the exact sequence of changes that happened in a project; we even record the time stamps and the exact contents of the changes, as we dispose of a fully reified model of changes. Put simply, we do not lose *anything* about the changes that concern a system. Replaying such IDE interactions is similar to Hassan and Holt's development replay approach, with the granularity necessary to also evaluate development-style prediction. Moreover, we provide more information to the predictors, allowing them to increase their accuracy. Since our interaction histories are replayable at will, we are able to systematically evaluate the effect of any change in the algorithm and the type of information it considers.

Our approach is based on our previous work on change-based software evolution (CBSE) [16]. CBSE has previously been used to support several reverse and forward engineering approaches [17], [18]. We implemented our approach as the Spyware tool platform [19], an IDE plugin that silently records and reifies all changes *as they happen*, and stores them in a change-based software repository.

Project	Duration (days)	Number of... classes	methods	sessions	changes	Predictions classes	methods
Spyware	1,095	697	11,797	496	23,227	6,966	12,937
Software Animator	62	605	2,633	133	15,723	3,229	8,867
Project X	98	673	3,908	125	5,513	2,100	3,981
Student project A	7	17	228	17	903	259	670
Student project B	7	35	340	19	1,595	524	1,174
Student project C	8	20	260	19	757	215	538
Student project D	12	15	142	17	511	137	296
Student project E	7	10	159	22	597	175	376
Student project F	7	50	454	22	1,326	425	946
Total					50,152	14,030	29,785

Table I
DEVELOPMENT HISTORIES IN THE BENCHMARK

A. Data Corpus

Our dataset contains the following systems:

- SpyWare, our prototype, monitored over a period of three years, constitutes the largest data set. The system has currently around 25,000 lines of code in ca. 700 classes. We recorded close to 25,000 changes so far.
- A Java project developed over 3 months, the Software Animator. In this case, we used our Java implementation of Spyware, an Eclipse plugin called EclipseEye [20].
- Six one-week student projects (projects A to F) with sizes ranging from 15 to 40 classes, with which we tested the accuracy of approaches on limited data. These development histories test whether an approach can adapt quickly at the beginning of a fast-evolving project.
- A Smalltalk project (Project X) authored by a professional Smalltalk developer, tracked during ca. 4 months.

The characteristics of each project are detailed in Table I, namely the duration and size of each project, in terms of classes, methods, number of changes and sessions. A development session represents continuous activity and is close to the granularity of a versioning system commit. We detect development sessions by comparing the time stamp of each change to the next: If two changes are separated by more than one hour, we start a new development session. We also report the number of times each change prediction algorithm was tested; in total, each change prediction algorithm was evaluated more than 40,000 times.

B. Benchmarking Procedure

Our benchmarking procedure is close to the development replay, only at a much finer level, in order to simulate as closely as possible the workflow of the developers as they built the systems for which we recorded the interactions.

Our benchmarking algorithm takes as parameters a change predictor and an interaction history, and returns lists of predictions and expected results. These are used at a later stage to compute the performance of the predictions. It is applicable both at the class level (predicting changes to classes) as a comparison point with the previous evaluations, and at the method level. Predicting changes at the method level gives more focused recommendations to the developer and is hence more useful.

The first unit the algorithm considers is the development session, equivalent in granularity to SCM transactions. We do not split the session into two sets as other approaches do. Instead, the already executed changes and the state of the system constitute the knowledge available to the predictor. The entities forming the expected set to predict are the targets of the upcoming changes, methods or classes. If predicting classes, changes to methods are considered to also change the class. The order of the entities is kept, and duplicates entities are removed. The predictor is tested before each single change and the results are stored for later evaluation. The change is subsequently executed to update the information available to the predictor.

During each run, we test the accuracy of a change prediction algorithm over a program's history, by processing each change one by one. We first ask the algorithm for its guess of what will change next, evaluate that guess compared to the actual change, and then pass that change to update its representation of the program's state and its evolution.

Some changes are directly submitted to the prediction engine without testing it first. They are still executed, since the engine must have an accurate representation of the program. These are (1) changes that create new entities, since one cannot predict anything for them, (2) repeated changes, i.e., if a change affects the same entity than the previous one, it is skipped, and (3) refactorings or other automated code transformations, since these are the result of automated tools and not the developer themselves.

C. Evaluating Prediction Performance

Evaluating the performance of the approaches requires careful attention. We need to define an evaluation procedure

Oracle at t_1: **A B E** B **D C** B B A D A A D C
Oracle at t_5: - - - - **D C B** B **A** D A A D C

Relevance of entities	A	B	C	D	E	F, G...
at t_1	1	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{5}}$	$\frac{1}{\sqrt{4}}$	$\frac{1}{\sqrt{3}}$	0
at t_5	$\frac{1}{\sqrt{4}}$	$\frac{1}{\sqrt{3}}$	$\frac{1}{\sqrt{2}}$	1	0	0

Table II
ASSIGNING RELEVANCES TO ENTITIES ACCORDING TO THE ORDER OF THEIR FIRST OCCURRENCES IN THE ORACLE (IN BOLD)

taking into account the nature of the task and the specificity of the information at our disposal.

A characteristic our data has over SCM data is a much stronger sense of ordering that we can exploit in the performance evaluation. The order of changes to entities as well as the richer structure of our data in general gives us the possibility of giving a higher importance (what is defined as *relevance* in information retrieval) to some entities with respect to others.

In the classical definitions of precision and recall, the relevance of an element is binary. In the case of transactions, the entity is either changed in it, or not. In addition, precision and recall do not take into account the rank recommendations occupy in a list. The fact that the most relevant entity is returned first or later in the list does not matter.

We use an alternative information retrieval performance measure called the cumulative gain [21]. Variants of the cumulative gain measure were used in official information retrieval benchmarks such as the ad-hoc track of INEX 2006 [22]. Cumulative Gain (CG) measures the performance of an IR system while taking into account how relevant a document is, and at which rank it is returned in a query. The relevance of the retrieved document is pondered by a *discount* coefficient modelling how far a user is willing to search down a list of recommendations. High discount factors simulate a user looking only at a few items in the list, while lower discount factors model a more persistent user. This variant of the CG is called Discounted Cumulative Gain, or DCG.

Finally, the CG measure takes into account variable relevances levels, i.e., the relevance of each document is a continuous value between 0 and 1, instead of a binary value.

Reporting the cumulative gain value at rank k gives an overall measure of the IR technique, accross all the ranks up to k. To compare several measures together, one can just compare them to the ideal measure, which returns the most relevant documents in the optimal order.

In the development case, the user wants to have a small list of entities he will need to change in the near future, to help him navigating to these entities. In this case, we set up the relevance of each entity as a function of how soon it will be changed, i.e., the very next entity to be changed will have a relevance of 1, the next one of $\frac{1}{\sqrt{2}}$, and so on until the last entity in the session is reached. In addition, we set up a discount factor with a high value as the developer is unwilling to look through a large list when he needs a fast navigation. We also limit the computation of the DCG (discounted cumulative gain) value up to the rank 9 as we model a small list of results in order to reflect the average processing capacities of humans [23].

This process is repeated for every individual test of the predictor, and the computed values are averaged. For comparison, they can be expressed as a ratio over the ideal measure which always returns the most relevent documents in the optimal order.

D. How to read the results

In the next section we measure the performance of a number of approaches using the previously presented benchmark.

For each of the projects we compute the cumulative gain for class and method predictions. The results are reported as ratios over the ideal cumulative gain for easy comparison between approaches. Instead of reporting the ratios as numbers between 0 and 1, we report values between 0 and 100 to ease readability. The results are computed for each project, and an average is reported in the end. Note that the average is weighted by the number of tests performed for each project.

IV. RESULTS

In this section we detail the evaluation of a number of change prediction approaches using our benchmark. We reproduced approaches presented in the literature and also implemented new approaches ourselves. In the case of reproduced approaches, we mention the eventual limitations of our reproduction and the assumptions we make. This is followed by the results of each approach and a brief discussion.

A. Association Rules Mining (ARM)

Description. This is the approach employed by Zimmermann et al. [3], and by Ying et al. [2]. The algorithm analyzes past transactions to infer rules over the patterns of changes in entities. These rules are of the type: "If A and B change, then C changes with them 60% of the time". When asked for a prediction, the algorithm looks in its rule base for rules whose antecedent corresponds to the entities that have recently changed, and proposes the consequents of the rules as predictions. These predictions are ordered by the amount of support each rule has, i.e., how often it was found valid in the past.

Like Zimmermann's approach, our version supports incremental updating of the dataset to better fit incremental development (instead of analysing the whole history at once). As the original approach uses SCM transactions, we

make the assumption that one session corresponds to one commit in the versioning system.

When processing each change in the session, it is added to the transaction that is being built. When looking for association rules, we use the context of the 5 preceding changes. We mine for rules with 1 to 5 antecedent entities, and return the ones with the highest support in the previous transactions in the history. Like Zimmermann, we only look for single-consequent rules.

Project	SW	SA	X	A–F	Avg
Classes	10.17	12.88	6.46	21.24	11.82
Methods	1.66	3.35	1.15	3.90	2.41

Table III
RESULTS OF ASSOCIATION RULES MINING

Results (Table III). The performance is low overall: The accuracy for methods is on average less than 5% of the overall accuracy. In the case of classes, the reduced amount of possibilities makes the approach perform better.

Variant: Immediate Rule Mining (IRM, Table IV). The main drawback of the approach is that it does not take into account changes in the current session. If entities are created during it, as is the case during active development, prediction based on previous transactions is impossible. To address this, we incrementally build a transaction containing the changes in the current session and mine it as well as the previous transactions. Maintaining this transaction allows for rules in which new entities can figure.

Project	SW	SA	X	A–F	Avg
Classes	21.04	21.37	25.80	30.21	22.99
Methods	9.70	11.31	10.27	11.45	10.48

Table IV
RESULTS OF IMMEDIATE RULES MINING

The results of this simple addition are shown in Table IV. The prediction accuracy at the class-level and the method-level are much higher: Improvements range from 50% (A–F, classes), to nearly tenfold (X, methods). Incrementally building the current session, and mining it allows us to quickly incorporate new entities which have been created in the current session, something that the default approach of Zimmermann does not support.

B. Degree of Interest (DOI)

Description. Mylyn maintains a degree-of-interest model [24] for entities which have been recently changed and viewed. The algorithm monitors the activity in the IDE and detects when the programmer interacts with elements by either viewing or changing them. Each time such an event occurs, the algorithm increases the interest value of the element who is the target of the interaction. This interest value decays over time if the element is not interacted with anymore: Each time an event occurs, the interest of all the other elements drops slightly. The algorithm in [6] mentions an interest increase of 1 for views, 2 for editions, and a decay value of 0.1. We implemented the same algorithm, with the following limitations:

- The original algorithm takes into account navigation data in addition to change data. Since we have recorded navigation data only on a fraction of the history, we do not consider it. We make the assumption that navigation data is not essential in predicting future change. Of course, one will probably navigate to the entity he wants to change before changing it, but this is hardly a useful recommendation.
- Another limitation is that more recent versions of the algorithm [6] maintain several degrees of interests based on manually delimited tasks. The tasks are then recalled by the developer manually. We do not consider separate tasks. The closest approximation of that for us is to assume that a task corresponds to a session, maintain a degree-of-interest model for each session, and reuse the one most related to the entity at hand.

Project	SW	SA	X	A–F	Avg
Classes	13.44	17.35	17.20	38.38	18.23
Methods	16.27	18.80	13.84	23.27	17.76

Table V
RESULTS OF DEGREE OF INTEREST

Results (Table V). At the class level, the degree of interest is less precise than IRM. At the method level, it has however a much higher accuracy. For finer-grained prediction, the interest model is more precise. Since the DOI in Mylyn is used to help development-style navigation, this finding is not surprising. However, agregating it to coarser entities is worse (comparatively).

Variant: DOI with navigation (DOI_N, Table VI). In the largest project we monitor, Spyware, we also recorded navigation information, so we were able to run the original version of the algorithm on this restricted set of data in order to check our assumption that navigation information was not essential. We indeed saw an improvement, of roughly 3.5% for classes, which is significant, and 1% for methods which is rather slight..

C. Coupling-based (CP)

Description. Briand et al. found that several coupling measures were good predictors of changes [9]. Indeed a method calling or being called by another indicates a strong relationship between the two entities. We define the set of

Project	SW	SA	X	A–F	Avg
Classes	16.75	-	-	-	16.75
Methods	17.30	-	-	-	17.30

Table VI
RESULTS OF DOI WITH NAVIGATION

methods being coupled with a method m as all the methods calling m and all the methods that m calls. We aggregate this measure at the class level by considering all the methods that a class c implements.

Our algorithm makes predictions based on the list of coupled entities with the last changing entity. These entities are ordered by the strength of their relationship in the case of classes. One limitation is that the Smalltalk systems do not have any type information, due to the dynamically typed nature of Smalltalk. Our version of the Java system also does not have type information. Thus a degree of imprecision is added in the case where several methods carry the same name.

Project	SW	SA	X	A–F	Avg
Classes	17.94	8.89	5.29	25.42	15.22
Methods	4.08	3.99	0.29	6.52	3.99

Table VII
RESULTS OF SENDER/IMPLEMENTOR COUPLING

Results (Table VII). The results are significantly higher than ARM, but lower than IRM and DOI, especially at the method level. The wide variations between SpyWare and project X lead us to think this approach is affected greatly by the working patterns of people.

Variant: Ordering methods by recent usage (CP_{Ord},Table VIII).

One major drawback of the approach is that no order can be defined at the method level, which significantly decreases the performance of the approach. To address this issue, we can consider the change history of the system to guide us. The set of methods coupled with a method m can be simply ordered by the date of their last change: The methods which changed more recently are put first. The results show a more consistent picture accross the board, with a higher method-level accuracy overall (+68%). For classes, the results are mixed: SpyWare experiences a great drop, whearas other projects show slight improvements or decreases in performance; On average, the performance drops by 50%. Overall, this ordering still results in a quite low performance.

Variant: Children, Parents, Siblings and Spouses (CP_{Fam}, Table IX, and CP_{OrdFam}, Table X. Another issue is that the set of entities considered may be too small. Saul et al. defined a structural recommendation algorithm which considered 4 sets of methods related to a given method m. The Parents are the methods which call m, while the Children are the methods that m calls. These two sets are included in our algorithm. However, the Siblings (the Children of the Parents of m, excluding m), and the Spouses (the Parents of the Children of m, excluding m) are not included in CP. Adding these indirectly coupled methods to our predictions, gives us a higher method-level accuracy, projects X and A-F being the clear winners. Ordering by time yields surprising results: Project SW and X suffer, while others are mostly unaffected. Once again, coupling-based approaches seem less stable, and harder to aggregate, than DOI and ARM-based approaches.

Project	SW	SA	X	A–F	Avg
Classes	7.29	10.11	5.17	24.10	9.96
Methods	7.39	6.83	0.73	9.71	6.82

Table VIII
RESULTS OF CHANGE-ORDERED COUPLING

Project	SW	SA	X	A–F	Avg
Classes	8.84	9.42	9.10	29.05	11.76
Methods	7.12	6.55	8.00	12.47	7.87

Table IX
RESULTS OF FAMILIAL COUPLING

Project	SW	SA	X	A–F	Avg
Classes	3.00	9.22	4.93	25.61	7.77
Methods	4.17	7.99	9.92	16.61	7.78

Table X
RESULTS OF ORDERED FAMILIAL COUPLING

D. HITS

Description. The HITS algorithm [25] is an algoritm used in web searching, similar to Google's PageRank [26]. It takes as input a graph of nodes linked by their (directed) relationships, and compute two metrics of interest for each node: A hub value and an authority value. Good hub nodes point to many good authority (also known as sinks) nodes, and good authorities are referred to by many good hubs. In the case of web searching, nodes are web pages, and edges are links between them. A authority in that case is a page linked to by many other pages (since many pages link to it, it is seen as an authority on its) subject, while a good hub links to many authorities. The HITS algorithm iteratively

computes the hub values based on the authority values, and the authority values based on the hub values, in several steps.

To adapt this algorithm to software, we need to define what are the nodes and the edges of the graph. Several variants can be defined. The entities returned by the prediction will then be the entities with the highest hub or authority values in the graph.

Variant: FRAN (HITS$_{FRAN}$, Table XI). FRAN is a method investigation algorithm proposed by Saul et al.. It uses HITS graph structure of the call graph between methods in either of the Parents, Children, Siblings and Spouses sets of the entity of interest. In our case, the entity of interest is the one that last changed. The best authorities are returned in all cases. Saul et al. mention that their approach is aimed at program investigation, where a larger number of matches is required, more than navigation assistance. Our results concurr, as its performance for development-style change prediction is low.

Project	SW	SA	X	A–F	Avg
Classes	4.54	6.69	4.74	26.11	8.00
Methods	1.21	2.35	2.65	3.86	2.09

Table XI
RESULTS OF HITS BASED ON FRAN

Variant: History-based (HITS$_{Changes}$, Table XII). In this version, graph nodes are classes and methods which have recently changed (i.e., involved in the last 50 changes to the system). The links between nodes are containment links (i.e., links from classes to methods), and links from entity to entity based on which entity changed before which other: Change history links chain methods and classes touched by successive changes, in chronological order. This version gives us much better results than other approaches: It is on average nearly twice as good as IRM for classes, and outperforms DOI for methods by more than 5%. Further, the performance is pretty stable accross the board. Note that we found best results by returning the best sinks (authorities), not the best hubs (average of hubs for classes: 33.77; methods: 21.54).

Project	SW	SA	X	A–F	Avg
Classes	40.02	38.68	37.33	48.11	40.46
Methods	21.63	27.07	17.69	24.91	23.20

Table XII
RESULTS OF HITS ON CHANGES

Variant: History and Structure-based (HITS$_{ChStruct}$, Table XIII). This variant overlays sender and implementors over the previous graph. If the results are good by themselves, they are disappointing overall, as they are roughly equivalent to HITS$_{Changes}$'s performance (decreasing for classes, and very slightly increasing for methods), despite making use of more information. As before, sinks yielded the best performance (average of hubs for classes: 29.11; methods: 21.06).

Project	SW	SA	X	A–F	Avg
Classes	36.84	38.48	35.36	44.38	38.05
Methods	21.75	27.12	18.02	25.45	23.39

Table XIII
RESULTS OF HITS ON CHANGES AND STRUCTURE

E. Reality Check (Recent Changes, RC)

Given the difference in performance between HITS$_{Changes}$ and HITS$_{ChStruct}$, we hypothesized that if simpler HITS graphs were performing best, then simple approaches might work best overall. We devised an approach based solely on recent changes: It proposes the n most recently changed entities, ordered by recency of changes.

Project	SW	SA	X	A–F	Avg
Classes	39.93	39.51	37.98	49.76	40.92
Methods	22.28	28.85	15.63	26.09	23.93

Table XIV
RESULTS OF RECENT CHANGES ONLY

Results (Table XIV). We were quite surprised to find this very basic approach be our best performer overall. It does not outperform HITS$_{Changes}$ by much, but considering its extreme simplicity, the performance is extremely convincing.

Variant: Introducing coupling-based ordering. We tried to alter the ordering of the recent changes based on whether they were part of the last changed method's family, but the only outcome was to slightly degrade the performance (prioritizing recently changed senders/implementors had a method-level of 20.54, instead of 23.93; also prioritizing siblings and spouses yielded a performance of 22.83).

F. Discussion of the results

We recall the average of each approach when predicting classes and methods in Table XV. Starred items (DOI$_N$) have not been run on all the projects, because of missing data.

Based on the results, we make the following observations:

Recent change information is key. Using recent change information gave us the best performing approaches (DOI, HITS on changes and recent changes), and considerably improved some. For example, the difference between ARM and IRM is due only to the presence of recent changes. In the

Approach	classes	methods
ARM	11.82	2.41
IRM	22.99	10.48
DOI	18.23	17.76
DOI_N*	16.75	17.30
CP	15.22	3.99
CP_{Ord}	9.96	6.82
CP_{Fam}	11.76	7.87
CP_{OrdFam}	7.77	7.78
$HITS_{FRAN}$	8.00	2.09
$HITS_{Changes}$	40.46	23.20
$HITS_{ChStruct}$	38.05	23.39
RC	40.92	23.93

Table XV
RESULTS OF ALL APPROACHES

same fashion, Hits approaches based on changes outperform those using structural information.

Coupling is unstable. Using coupling information yielded unstable performance. Some of our projects reacted very differently from others: Some were affected positively by ordering of time, others negatively. We are unsure of the reason of the differences, but hypothesize that the way people interact with their code affects this.

In the same fashion, aggregating coupling and ordering information from methods to classes gave mixed results. It worked well for some projects, but not for others.

HITS performs best with sinks. HITS-based algorithms use return two values, a sink and a hub metric; returning the best sinks consistently outperformed the best hubs.

Simple approaches work best – so far. Our best-performing approach is simple: It simply recommends the most recently changed entities. It matches the workflow of coding pretty well, as one often changes code incrementally and hence needs to return to entities he or she is working on.

We are still hopeful to improve these results further, and want to investigate two leads. The first is to combine approaches by merging their results, and find out empirically which merging strategy, and which combination of approaches yield best results. The second is to further investigate context-aware strategies, able to selectively look farther in the past history of the project. Mylyn's DOI has been extended to include several task contexts in this manner [6]. We suspect we will need to update our benchmark procedure in order to select cases where this behavior is useful.

V. THREATS TO VALIDITY

There are a number threats to the external validity of our approach:

Not all approaches were reproduced. We did not reproduce the Navtracks approach as it relies only on navigation data, which we do not have. Ying and Shirabad's approaches are very close to Zimmermann's association rule mining. DeLine et al.'s Teamtrack is based on a DOI and is as such close to the Mylyn DOI. Kagdi's approach was not fully described at this time of writing. Finally, we chose only one coupling measure to reproduce, while many exist. The one we chose was the one best adapted to our systems as PIM takes polymorphism into account. In Briand et al.'s study, PIM was one the metrics with the best correlation with actual changes.

Size of the dataset. Our dataset is fairly small, if not tiny compared to the ones available with versioning system data. With time, we will incorporate more data in our benchmark in order to be more comprehensive. On the other hand, our benchmark is already larger than the ones used in previous studies by Briand, Wilkie or Tsantalis. Their evaluations were done on one or two systems, on a small number of transactions.

Generalizability. We do not claim that our results are generalizable. They however constitute good initial results in the case of rapidly developing systems since all nine projects fit that description with varying degrees of size. In addition, some of the results we found were in line with results found by other researchers. Hassan and Holt found that coupling-based approaches are less precise than history based approaches, and so do we.

VI. CONCLUSION

In this paper we presented a benchmark to repeatedly evaluate the accuracy of change-prediction approaches. It is unique since it is based on recording the history of programs in a realistic setting by monitoring the programmers as they build their systems.

By replaying the change history at the level of individual changes in a development session, we feed more accurate data to the change prediction algorithms. This allows evaluation of these change prediction approaches to proceed without necessarily involving a controlled experiment which is more expensive to perform, and harder, if not impossible to reproduce.

As noted by Sim et al., benchmarks also need to evolve [27]; this our case as well. Our benchmark is still relatively small, so we need to integrate the histories of other programs. Additional data in the form of navigation and typing information data is also needed to accomodate a greater variety of approaches. Since our data includes also the actual changes performed, and not only the entities, predicting the actual change contents would be an interesting, albeit ambitious, extension.

Using our benchmark, we compared several existing approaches that we replicated (association rules mining, degree of interest, and coupling-based impact analysis), with other approaches we developed (variations of association rules mining, approaches using the HITS algorithm, and a simple

approach recommending recent past changes). Our results show that of the approaches we tested, the ones with the most accuracy were the ones based on recent changes: the simplest one was the best performer overall, by a slight margin. We plan to investigate how to improve results further by merging the recommendations of several approaches, and by specifically evaluating the cases when an entity that has been changed farther in the past needs to be changed again.

REFERENCES

[1] G. M. Weinberg, *The Psychology of Computer Programming*, silver anniversary edition ed. Dorset House, 1998.

[2] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll, "Predicting source code changes by mining change history," *Transactions on Software Engineering*, vol. 30, no. 9, pp. 573–586, 2004.

[3] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *ICSE*. IEEE Computer Society, 2004, pp. 563–572.

[4] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proceedings International Conference on Software Maintenance (ICSM '98)*. Los Alamitos CA: IEEE Computer Society Press, 1998, pp. 190–198.

[5] J. Singer, R. Elves, and M.-A. Storey, "Navtracks: Supporting navigation in software maintenance," in *Proceedings of the 21st International Conference on Software Maintenance (ICSM 2005)*. IEEE Computer Society, sep 2005, pp. 325–335.

[6] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proceedings of SIGSOFT FSE 2006*, 2006, pp. 1–11.

[7] D. Röthlisberger, O. Nierstrasz, S. Ducasse, D. Pollet, and R. Robbes, "Supporting task-oriented navigation in ides with configurable heatmaps," in *ICPC*, 2009, pp. 253–257.

[8] J. Sayyad-Shirabad, T. Lethbridge, and S. Matwin, "Mining the maintenance history of a legacy software system," in *ICSM*. IEEE Computer Society, 2003, pp. 95–104.

[9] L. C. Briand, J. Wüst, and H. Lounis, "Using coupling measurement for impact analysis in object-oriented systems," in *ICSM*, 1999, pp. 475–482.

[10] F. G. Wilkie and B. A. Kitchenham, "Coupling measures and change ripples in c++ application software," *Journal of Systems and Software*, vol. 52, no. 2-3, pp. 157–164, 2000.

[11] A. E. Hassan and R. C. Holt, "Replaying development history to assess the effectiveness of change propagation tools," *Empirical Software Engineering*, vol. 11, no. 3, pp. 335–367, 2006.

[12] H. H. Kagdi, "Improving change prediction with fine-grained source code mining," in *ASE*, R. E. K. Stirewalt, A. Egyed, and B. Fischer, Eds. ACM, 2007, pp. 559–562.

[13] R. Robbes and M. Lanza, "Versioning systems for evolution research," in *Proceedings of IWPSE 2005 (8th International Workshop on Principles of Software Evolution)*. IEEE CS Press, 2005, pp. 155–164.

[14] R. DeLine, M. Czerwinski, and G. G. Robertson, "Easing program comprehension by sharing navigation data," in *VL/HCC*. IEEE Computer Society, 2005, pp. 241–248.

[15] J. Lung, J. Aranda, S. M. Easterbrook, and G. V. Wilson, "On the difficulty of replicating human subjects studies in software engineering," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 191–200.

[16] R. Robbes, "Of change and software," Ph.D. dissertation, University of Lugano, December 2008. [Online]. Available: http://www.inf.unisi.ch/phd/robbes/OfChangeAndSoftware.pdf

[17] R. Robbes and M. Lanza, "Example-based program transformation," in *Proceedings of MODELS 2008 (11th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems)*. ACM Press, 2008, pp. 174–188.

[18] R. Robbes and M. Lanza, "How program history can improve code completion," in *Proceedings of ASE 2008 (23rd ACM/IEEE International Conference on Automated Software Engineering)*. ACM Press, 2008, pp. 317–326.

[19] R. Robbes and M. Lanza, "Spyware: A change-aware development toolset," in *Proceedings of ICSE 2008 (30th International Conference in Software Engineering)*, 2008, pp. 847–850.

[20] Y. Sharon, "Eclipseye - spying on eclipse," University of Lugano, Bachelor's thesis, Jun. 2007.

[21] K. Järvelin and J. Kekäläinen, "Cumulated gain-based evaluation of ir techniques," *ACM Transactions on Information Systems*, vol. 20, pp. 422–446, 2002.

[22] M. Lalmas, G. Kazai, J. Kamps, J. Pehcevski, B. Piwowarski, and S. Robertson, "Inex 2006 evaluation measures," in *INEX*, 2006, pp. 20–34.

[23] G. A. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information," *The Psychological Review*, vol. 63, pp. 81–97, 1956. [Online]. Available: http://users.ecs.soton.ac.uk/~{}harnad/Papers/Py104/Miller/miller.html

[24] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for ides." in *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005)*, 2005, pp. 159–168.

[25] J. Kleinberg, "Authoritative sources in a hyperlinked environment," IBM, Tech. Rep., May 1997.

[26] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," Computer Science Department, Stanford University, Tech. Rep., 1998.

[27] S. E. Sim, S. M. Easterbrook, and R. C. Holt, "Using benchmarking to advance research: A challenge to software engineering," in *ICSE*. IEEE Computer Society, 2003, pp. 74–83.