A Benchmark for Change Prediction

Romain Robbes, Michele Lanza
REVEAL @ Faculty of Informatics – University of Lugano, Switzerland
romain.robbes@lu.unisi.ch, michele.lanza@unisi.ch
Damien Pollet
University of Lille
damien.pollet@gmail.com

University of Lugano
Faculty of Informatics
Technical Report No. 2008/06
October 2008

Abstract

The goal of change prediction is to help 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 Subversion. However, as these data sources are not very accurate, they do not provide a valid basis for an objective evaluation of change prediction approaches.

We propose a benchmark for an objective 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 to evaluate several change prediction approaches, both our own and from the literature, and report on the results. Our results show that using fine-grained change data significantly improves the overall accuracy of change prediction approaches.

1 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 slow paradigm shift taking place, with people slowly departing from the notion that source code is equivalent to text writing [25]. Despite the many vi and emacs fundamentalists, developers are going 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 increased "assistance" to programmers [28]. In his vision of future IDEs, developers in the future will be supported 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. Depending on the development phase, change prediction has different usages.

For *software maintenance*, change predictors recommend changes to entities that may not be obvious [29, 27]. In a software system, there often exist implicit or indirect relationships between entities [3]. 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.

At the current stage, it is difficult to assess whether a predictor performs better than another one, i.e., 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 an *ad hoc* benchmark to measure improvements, as it can be reproduced and is cheap to run, as opposed to user studies. However, during forward engineering, when the system changes at a quick pace, this 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, no satisfactory approach has been proposed, and tools adapted to the forward engineering use case are assessed through comparative studies involving developers [9, 23], a labor-intensive, error-prone and imprecise process.

Having a benchmark for the second case would however be a significant help to design change recommenders. Sim *et al.* documented the effect benchmarks have on scientific communities [22]. Since they allow communities to easily compare their results, they tend to accelerate the research. Their advice for the software engineering community was to adopt benchmark as a practice whenever it was possible.

We present a unifying benchmark for both kinds of change prediction. The benchmark consists of a number of fine-grained development histories, where we recorded *each* change that happened to the systems while they were developed. Our detailed histories are thus unaffected by the change size problem. In a nutshell, our benchmark allows us to reproduceably *replay* entire development histories, thus providing (close to) real life benchmark 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.

Structure of the paper. Section 2 describes various change prediction approaches existing in the literature in the two change prediction styles. Section 3 justifies and presents our benchmark for change prediction approaches. Section 4 details the approaches we evaluated with our benchmark and presents the benchmark results, which we discuss in Section 5, before concluding in Section 6.

2 Related Work

Several change prediction heuristics have been proposed. We find 3 major trends: *Historical* approaches, *IDE-based* approaches, and approaches relying on *Impact Analysis*.

Historical Approaches use the history of an SCM system to predict changes, primarily in a maintenance setting.

Zimmerman *et al.* [29] 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. It is indeed impossible to predict new features from the history. 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 [27]. 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 [20], but stayed at the file level.

Girba also detected co-change patterns [4]. Thanks to his precise evolution model, he also qualified them, with qualifiers such as Shotgun Surgery, Parallel Inheritance or Parallel Semantics. He also proposed the *Yesterday's Weather* approach, which postulates that future changes will take place where the system just changed [5].

IDE-based approaches. The goal of short-term, IDE-based 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.

Mylyn [9] 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, and 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. It was shown that using Mylyn, developers spent more time editing code, and less time looking for places to edit.

Navtracks [23] and Teamtracks [2] both record navigation events to ease navigation of future users, and are geared towards maintainance activities. Teamtracks also features a DOI model. Navtrack's recommendations are at the file level. Teamtracks was validated with user studies, while Navtracks was validated both with a user study and also by recording the navigation of users and evaluating how often Navtracks would correctly predict their navigation paths (around 35% of the recommendations were correct).

SCM-validated Impact Analysis Approaches. Impact analysis has been performed using a variety

of techniques; we only comment on a few. Briand *et al.* [1] evaluated the effectiveness of coupling measurements to predict ripple effect changes on a 90 classes system. 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 [26] 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. Also Kagdi proposed a hybrid approach merging impact analysis techniques with historical techniques [7], but no results have been published yet.

Hassan and Holt have proposed an approach based on replaying the development history of projects based on their versioning system archives[6]. They 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. Similar to Zimmermann *et al.*, they observed that the GCC project has different results and hypothesize this is due to the project being in maintainance mode.

Tsantalis *et al.* change prediction approach was validated on two systems. One has 58 classes and 13 versions, while the other has 169 classes and 9 versions [24].

Project	Duration (days)	Classes	Methods	Sessions	Changes	Predictions Classes	Predictions Methods	Predictions Classes (Init)	Predictions Methods (Init)
Spyware	1095	697	11797	496	23227	6966	12937	4937	6246
Software Animator	62	605	2633	133	15723	3229	8867	1784	2249
Project X	98	673	3908	125	5513	2100	3981	1424	1743
Project A	7	17	228	17	903	259	670	126	236
Project B	7	35	340	19	1595	524	1174	210	298
Project C	8	20	260	19	757	215	538	151	251
Project D	12	15	142	17	511	137	296	122	156
Project E	7	10	159	22	597	175	376	148	238
Project F	7	50	454	22	1326	425	946	258	369
Total	-	-	-	-	50152	14030	29785	9160	11786

Table 1. Development histories in the benchmark.

3 A Benchmark for Change Prediction

In Section 2 we have listed a number of change prediction approaches that have been evaluated with data obtained either by mining the repositories of software configuration management (SCM) systems, such as CVS or SubVersion, or by tracking what happens within the IDE as a developer is programming. We argue that such evaluations suffer from two major problems:

1. *SCM data is inaccurate.* The data obtained by mining SCM repositories is potentially inaccurate: Some SCM transactions represent patch applications rather than developments and cannot be used by change predictions. Other transactions are simply too large to extract useful information. Also, in an SCM system transaction, there is no way to know which entity was changed first, *i.e.,* the chronological order the changes is lost.

2. *IDE data is shallow.* Approaches such as Mylyn, Navtracks and Teamtracks 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. They thus only have shallow data at their disposition, and rely on human-controlled experiments to assess their performance. While these experiments can be used to validate the overall effectiveness of a tool, they are very expensive to perform, are not suited for incremental improvements of the tools, and most of all the experiments are not repeatable.

By contrast, we propose an 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.

Our approach, named change-based software evolution (CBSE)[18] has previosuly been used to support program comprehension[14], and forward engineering [15, 16]. We implemented our approach as the Spyware tool platform [17], an IDE plugin that records and reifies all changes *as they happen*, and stores them in a change-based software repository. Of note, our recording is non-instrusive, Spyware silently records the data without disturbing the user.

3.1 Data Corpus

We selected the following development histories as benchmark data:

- 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[21], which however does not support the recording of usage histories.
- Nine one-week small student projects with sizes ranging from 15 to 40 classes, with which we tested the accuracy of approaches on limited data. These development histories test wether an approach can adapt quickly at the beginning of a fast-evolving project.
- A professional Smalltalk project tracked ca. 4 months.

The characteristics of each project are detailed in Table 1, namely the duration and size of each project (in term of classes, methods, number of changes and sessions), as well as the number of times the predictor was tested for each project, in four categories: overall class prediction, overall method prediction, and class and method prediction at the start of sessions only (this last point is explained in Section 3.3).

3.2 Benchmarking Procedure

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 asking it to guess the changes first. They are still processed, 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.

Input: $History$: Change history used
$Predictor$: Change predictor to test
Output: $Results$: benchmark results

```
Results = makeResultSet();
foreach Session S in ChangeHistory do
    storeSessionInfo(S, result);
    testableChanges = filterTestableChanges(S);  foreach Change ch in S do
        if includes(testableChanges,ch) then
            predictions = predict(Predictor);
            nbPred = size(predictions);
            oracle = nextElements(testableChanges, nbPred);
            storeResult(results, predictions, oracle);
        end
        processChange(Predictor, ch);
    end
    return Results
end
```

Algorithm 1: Benchmark result collection

The pseudo-code of our algorithm is shown in Algorithm 1. It runs on two levels at once, asking the predictor to predict the next changing class and the next changing method. When a method changes, the predictor is first asked to guess the class the method belongs to. Then it is asked to guess the actual method. When a class definition changes, the predictor is only tested at the class level. This algorithm does not evaluate the results, but merely stores them along with the actual next changing entities, allowing them to be evaluated later on in a variety of ways. To allow a finer characterization of the results, we also store several session metrics, namely the number of (a) changes in the session, (b) methods additions, (c) methods modifications (changes to an already existing method), (d) class additions, and (e) class modifications.

3.3 Evaluating Prediction Performance

With these results stored, we can evaluate them in a variety of ways. All of them share the same performance measurement, but applied to a different set of predictions. Given a set of predictions, we use Algorithm 2 to return an accuracy score.

Given a list of n predictions, the algorithm compares them to the next n entities that changed, and sets the accuracy of the algorithm as the fraction of correct predictions over the number of predictions.

Input: $Results$: Benchmark results
$Depth$: Number of predictions to evaluate
Output: $Score$: Accuracy Score

$accuracy$ = 0;
$attempts$ = size($results$);
foreach *Attempt* att *in* $attemps$ **do**
    $predictions$ = getPredictions(att, $Depth$);
    $oracles$ = getOracles($oracles$, $Depth$);
    $predicted$ = $predictions \cap oracles$;
    $accuracy$ = $accuracy$ + (size($predicted$) / size($predictions$));
end
$Score$ = $accuracy$ / $attempts$;
return $Score$

Algorithm 2: Benchmark result evaluation

In the case where less than n entities changed afterwards (m), only the m first predictions are taken into account.

Accuracy vs. Prediction and Recall. Change prediction approaches that use SCM data are often evaluated using precision and recall. We found however that our data does not fit naturally with such measurements, because while recorded changes are sequential, some development actions can be performed in any order to a certain extent: If a developer has to change three methods A, B, and C, he can do so in any order he wants. To account for this parallelism, we do not just test for the prediction of the next change, but for the immediate sequence changes with length n. Defining precision and recall for the prediction set and the set of the actual changes would make both measures have the same value. This does not fit the normal precision and recall measures which vary in inverse proportion to each other. Intuitively, approaches with a high recall make a greater number of predictions, while approaches with a higher precision make less predictions, which are more accurate. Since we fix the number of predictions to a certain size n, we use one single accuracy measure.

We measure the following types of accuracy:

- Coarse-grained accuracy (C) measures the ability to predict the classes where changes will occur.
- Fine-grained accuracy (M) measures the ability to predict the methods where changes will occur.
- Initial accuracy (I) measures how well a predictor adapts itself to a changing context both for classes and methods. To evaluate how fast a change predictor reacts to a changing context, we measure its accuracy of each predictor on only the first changes of each session. These feature the highest probability that a new feature is started or continued. We thus measure the accuracy of the 20 first changes of each session.

How the read the results. In the next section we measure the accuracy of a number of approaches using the previously presented benchmark. We present the results in tables following the format of the sample Table 2.

Project	C5	C7	C9	M5	M7	M9	CI7	MI7
SW	20	42	32	23	32	98	67	45
A-F	30	42	32	23	32	98	67	45
SA	40	42	32	23	32	98	67	45
X	50	42	32	23	32	98	67	45
AVG	37	42	32	23	32	98	67	45

Table 2. Sample results for an algorithm

For each of the projects (occupying the rows) we compute the coarse-grained (C5, C7, C9), the fine-grained (M5, M7, M9) and the initial accuracy for classes (CI7) and for methods (MI7). The digits (5,7,9) indicate the length of the prediction and validation set, *e.g.,* M7 means that we measure the accuracy of change prediction for a sequence of method changes of length 7.

How are the numbers to be understood? For example, in the C5 column we measure the coarse-grained accuracy of one of the approaches. The '20' in the Spyware (SW) row means that when it comes to predicting the next 5 classes that will be changed, the fictive predictor evaluated in Table 2 is guessing correctly in 20% of the cases. In the case of the small student projects (row A-F) the values indicate how the predictors perform on very small sets of data. The Software Animator (SA) row indicates how the predictors perform on Java systems, while the second last row (X) indicates how the predictors perform on a system built around a large web framework. The bottom row contains the weighted average value accuracy for each type of prediction. The average is weighted with the number of changes of each of the benchmark systems. This means that the longest history, the one of Spyware, plays a major role, and that one should not expect arithmetic averages in the last row.

4 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 now 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.

We start with a few general remarks. First, the larger the number of matches considered, the higher the accuracy. This is not surprising. One must however limit the number of entities proposed, since proposing too many entities is useless. One can always have 100% accuracy by proposing all the entities in the project. This is why we limited ourselves to 7+2 entities, thus keeping a shortlist of entities, these having still a reasonable probability of being the next ones. This number is estimated to be the number of items that humans can keep in short-term memory [12].

Also, all the algorithms follow roughly the same trends accross projects. The smaller projects (A-F) have a higher accuracy, which is to be expected since there are less entities to choose from, hence a higher probability to pick the correct ones. The software animator (SA) project has a higher accuracy than SpyWare (SW), since it is also smaller. The project with the least accuracy overall is project X. Its development is constituted of a variety of smaller tasks and features frequent switching between these tasks. These parts are loosely related, hence the history of the project is only partially useful.

4.1 Association Rules Mining

Description. This is the approach employed by Zimmermann *et al.* [29], and by Ying *et al.* [27]. Like Zimmermann's approach, our version supports incremental updating of the dataset to better fit incremental development. The alternative would be to analyse all the history at once, using two-thirds of the data as a training set to predict the other third. This does however not fit a real-life setting. As the 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 in Zimmermann's approach, we only look for single-consequent rules.

Project	C5	C7	C9	M5	M7	M9	CI7	MI7
SW	13.0	15.5	17.5	2.7	2.9	3.1	17.2	4.1
A-F	27.3	32.0	37.1	3.3	3.7	4.0	34.7	5.5
SA	16.1	19.3	21.9	4.5	5.4	6.2	21.7	6.4
X	6.0	7.4	8.4	2.1	2.2	2.2	7.9	3.7
AVG	14.4	17.2	19.6	3.1	3.5	3.9	18.5	4.6

Table 3. Results for Association Rules Mining

Results. Association rule mining serves as our baseline. As Table 3 shows, the results are relatively accurate for class-level predictions, but much lower for method-level predictions. The results are in the range of those reported by Zimmermann *et al.* [29]. They cite that for the GCC case study which was under active development, the precision of their method was only around 4%. Our results for method-level accuracy are in the low single-digit range as well.

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.

Project	C5	C7	C9	M5	M7	M9	CI7	MI7
SW	30.0	36.1	40.8	13.7	16.2	18.4	40.0	23.1
A-F	39.7	45.6	52.2	12.4	14.9	16.6	50.6	23.4
SA	28.0	34.1	39.5	14.2	17.7	20.9	39.4	24.0
X	24.4	29.6	33.7	14.3	17.3	20.2	31.5	31.1
AVG	29.9	35.8	40.8	13.7	16.6	19.0	39.7	24.5

Table 4. Results for Enhanced Association Rules Mining

The results of this simple addition are shown in Table 4. The prediction accuracy at the class-level and the method-level is higher, but the method-level acccuracy is much higher. 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. Of note, the algorithm is more precise at the beginning of the session than at the end, because the current session has less entities to propose at the beginning of the session. Towards the end of the session, there are more possible proposals, hence the approach loses its accuracy. In the following, we compare other approaches with enhanced association rule mining, as it is a fairer comparison since it takes into account entities created during the session.

4.2 Degree of Interest

Description. Mylyn maintains a degree-of-interest model [8] for entities which have been recently changed and edited. 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 [9] maintain several degrees of interests based on manually delimited tasks. The tasks are then recalled by the developer. 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	C5	C7	C9	M5	M7	M9	CI7	MI7
SW	16.1	21.0	25.7	10.2	12.8	14.8	20.1	12.5
A-F	52.0	60.2	67.0	16.6	20.7	23.3	57.8	20.2
SA	22.4	29.8	35.5	21.2	26.9	31.0	29.7	25.2
X	15.5	19.5	22.0	6.1	7.4	8.2	17.1	6.5
AVG	21.9	27.6	32.5	13.2	16.6	19.1	25.7	14.9

Table 5. Results for Degreee of Interest

Results. We expected the degree of interest to perform quite well and found the results to be below our expectations. At the class level, it is less precise than association-rule mining. At the method level, it has roughly the same accuracy. The accuracy drops sharply with project X, whose development involved continuous task switching. Since the degree-of-interest needs time to adapt to changing conditions, such sharp changes lowers its performance. Indeed, the best accuracy is attained when the algorithm has more time to adapt. One indicator of this is that the algorithm's accuracy at the beginning of the session is lower than the average accuracy. The algorithm also performs well to predict classes in projects A-F, since their size is limited. It nevertheless shows limitations on the same project to predict methods given the very short-term nature of the projects (only a week).

4.3 Coupling-based

Description. Briand *et al.* found that several coupling measures were good predictors of changes [1].

We chose to run our measure with the PIM coupling metric, one of the best predictors found by Briand *et al.*. PIM is a count of the number of methods invocations of an entity A to an entity B. In the case of classes, this measure is aggregated between all the methods in the corresponding two classes.

To work well, the coupling-based approach needs to have access to all the code base of the system and the call relationships in it. With respect to this requirement, our benchmark suffers from a number of limitations:

- We could not run this test on project SA, since our Java parser is currently not fine-grained enough to parse method calls.
- We also do not include the results of Project X. Unfortunately, project X was already under development when we started recording its evolution, which would make the comparison unfair.
- One last limitation is that the Smalltalk systems do not have any type information, due to the dynamically typed nature of Smalltalk. While we can make use of a type inference engine to infer a certain portion of the types, it does make our PIM measure slightly less accurate.

Project	C5	C7	C9	M5	M7	M9	CI7	MI7
SW	21.0	26.2	30.6	9.7	11.8	13.3	25.3	11.2
A-F	21.3	28.0	34.0	10.8	14.1	16.6	29.9	15.7
SA	-	-	-	-	-	-	-	-
X	-	-	-	-	-	-	-	-
AVG*	21.0	26.6	31.3	9.9	12.3	14.0	26.1	12.1

Table 6. Results for Coupling with PIM

Results. As we see in Table 6, comparing with the other approaches is difficult since part of the data is missing. On the available data, we see that the coupling approach performs worse at the method level than Association Rules Mining and Degree of Interest. At the class level, results are less clear: The approach performs worse than association rules, while the degree of interest performs significantly better on projects A-F, but is outperformed at the class level.

Overall, we share the conclusions of Hassan and Holt [6]: Coupling approaches have a lower accuracy than history-based ones such as association rule mining. The comparison with degree of interest somewhat confirms this, although it was outperformed in one instance.

4.4 Association Rules with Time Coupling

Description. In previous work [19] we observed that a fine-grained measure of logical coupling was able to predict logical coupling with less data. We therefore used this coupling measurement instead of the classical one to see if it was able to better predict changes with association-rule mining. When mining

for rules, instead of simply couting the occurrences of each rule in the history, we factor a measure of time coupling. Time coupling measures how closely in a session two entities changed. Entities changing very closely together will have a higher time coupling value than two entities changing at the beginning and at the end of a development session.

Project	C5	C7	C9	M5	M7	M9	CI7	MI7
SW	30.1	36.0	40.6	13.2	16.0	18.3	38.0	21.4
A-F	37.0	43.6	50.8	14.6	17.3	19.3	45.7	23.3
SA	25.6	31.0	35.7	17.2	20.7	23.7	33.0	23.1
X	23.8	29.0	33.2	10.8	14.6	17.6	29.9	25.9
AVG	29.0	34.7	39.7	14.0	17.2	19.7	36.6	22.6

Table 7. Results for Association Rules with Time Coupling

Results. As we see in Table 7, our results are mixed. The prediction accuracy is slightly lower that Enhanced Association Rules Mining for class-level predictions, and slightly better for method-level predictions, each time by around one percentage point. It is encouraging that the method prediction is increased, since it is arguably the most useful measure: Precise indications are better than coarser ones. We hope to improve this alternative coupling measure in the future.

4.5 HITS

Description. The HITS algorithm [10] is similar to Google's PageRank [11]. It gives a hub and an authority (or sink) value to all nodes in a directed graph. Good hub nodes (or web pages) point to many good authority nodes, and good authorities are referred to by many good hubs. While PageRank works on hyperlinked web pages, we build a graph from classes and methods, linking them according to containment and message sends. The predictor has two variants, respectively recommending the best hubs or the best sinks, in the graph generated from the most recent changes. We maintain this graph for the entities and the calls defined in the 50 latest changes.

Project	C5	C7	C9	M5	M7	M9	CI7	MI7
SW	40.8	48.8	55.8	15.9	17.1	17.9	48.3	17.2
A-F	44.7	55.7	65.1	20.9	23.5	25.2	57.6	22.6
SA	55.4	71.5	85.7	31.3	35.6	38.9	73.6	33.0
X	30.4	33.7	36.7	7.7	8.4	8.8	32.4	8.6
AVG	43.1	52.6	61.0	19.2	21.4	22.8	51.8	19.6

Table 8. Results for Hits, best hubs

Results. The HITS algorithm proved to be the highest performer overall since it has a significantly higher method-level accuracy overall. We tested two variants, the first one returning the best hubs in the HITS graph, and the second returning the best sinks. As with the degree-of-interest, HITS tends to be more precise towards the end of a session. We need to investigate ways to make the algorithm adapt faster to new contexts.

Project	C5	C7	C9	M5	M7	M9	CI7	MI7
SW	50.0	55.9	61.1	16.7	17.9	18.6	55.8	17.7
A-F	57.6	66.0	74.6	22.3	24.9	26.7	65.8	24.0
SA	60.6	74.6	87.5	32.1	36.2	39.0	76.2	33.5
X	34.0	38.1	40.8	8.0	8.6	9.0	37.3	8.5
AVG	51.0	58.8	65.8	20.1	22.1	23.4	58.0	20.1

Table 9. Results for Hits, best sinks

4.6 Merging Approaches

Description. Kagdi *et al.* advocated merging history based approaches and impact analysis based approaches [7], arguing that combining the strong points of several approaches can yield even more benefits. This strategy was successfully used by Poshyvanyk *et al.* for feature location [13]. We tried the following merging strategies:

Top: Returning the top picks of the combined approaches. This approaches assumes that the first choices of an approach are the most probable ones. Given two predictors A and B, it returns A's first pick, then B's first pick, then A's second pick, *etc.*.

Common: Returning entities nominated by several approaches, followed by the remaining picks from the first approach. This approach assumes that the first predictor is the most accurate, and that the other predictors are supporting it. The approach returns all the predictions in common between at least two predictors, followed by the remaining predictions of the first predictor.

Rank addition: This approache favors low ranks and entities nominated several times. An interest value is computed for each prediction. Each time the prediction is encountered, its interest is increased by a value propartional to its rank in the prediction list it is in. If it is encountered several times, its interest value will thus be higher. The predictions are then sorted by their interest value.

Predictor 1	Predictor 2	Strategy	Score	Increase
Hits-Sinks	Coupling	Common	40.6	+0.1
Hits-Hubs	Interest	Common	37.8	+0.8
Hits-Hubs	Coupling	Common	37.3	+0.3
Rule Mining	Interest	Common	27.4	+1.2
Rule Mining	Time Coupling	Top	27.3	+1.1
Rule Mining	Coupling	Rank	26.3	+0.1
Time Coupling	Coupling	Rank	26.1	+0.2
Interest	Coupling	Rank	22.6	+0.5

Table 10. Results when merging two prediction approaches

Results. We tried these merging approaches on each possible combination of two and three predictors. In the following table we report on the successful ones, where the accuracy of the merging was greater than the best of the merged approaches. If several merging strategies were succesful with the same pair

of predictors, we only report the best performing one. We only report one accuracy figure, which is the average of C7 and M7.

We see that merging is successful in some cases, although the effect is limited. The predictions based on the HITS algorithm see some improvement only with the "Common" strategy, which favors the first predictor. This is to be expected, since these algorithms are significantly more accurate than the other ones. Systematically merging their top candidates with the ones of a less accurate approach automatically decrease their efficiency.

With other predictors, the other merging strategies become more viable, although the Top strategy appears only once. This strategy is the only one not rewarding the presence of common recommendations in the two predictors. In that case, merging the two strategies with the Rank strategy yielded an improvement of 0.9% instead of 1.1%. Overall, merging strategies favoring common predictions work best.

Perhaps the most important fact is that Coupling appears in five of the eight successful merging. This supports the idea that coupling based on the structure of the system proposes different predictions than history-based ones. Our result provide initial support to Kagdi's proposition of merging impact-analysis approaches (some of them using coupling measurements) with history-based approaches.

Surprisingly, merging three predictors instead of two yielded few benefits. Only in one instance was it better than merging two. Merging Association Rule Mining, Degree of Interest and Coupling had a score of 27.9, a 0.5% increase over merging only Association Rule Mining and Degree of Interest. Of note, only Rank and Common were successful in merging three predictors. The higher the number of predictors, the more important it is to favor common predictions.

Predictor	C5	C7	C9	M5	M7	M9	CI7	MI7	O7	MO7
Association Rules Mining	14.4	17.2	19.6	3.1	3.5	3.9	18.5	4.6	10.35	8.06
Enhanced Association Rules Mining	29.9	35.8	40.8	13.7	16.6	19.0	39.7	24.5	26.20	23.00
Degree of Interest	21.9	27.6	32.5	13.2	16.6	19.1	25.7	14.9	22.10	20.26
Coupling-based*	21.0	26.6	31.3	9.9	12.3	14.0	26.1	12.1	19.45	17.06
Association Rules Mining & Time Coupling	29.0	34.7	39.7	14.0	17.2	19.7	36.6	22.6	25.95	23.03
Hits, best Hubs	43.1	52.6	61.0	19.2	21.4	22.8	51.8	19.6	37.00	31.80
Hits, best Sinks	51.0	58.8	65.8	20.1	22.1	23.4	58.0	20.1	40.45	34.33

Table 11. Comprehensive results for each predictor

4.7 Discussion of the results

The results of all the predictors are summed up in Table 11. Note that the coupling results were not run on all the projects, and should as such be taken with a grain of salt. The last two columns represent an overview value for each predictor: O7 is the average of C7 and M7, while MO7 is another average of C7 and M7, favorizing method accuracy (C7 counts for a third and M7 for the two remaining thirds).

(Enhanced) Association Rules Mining. Taking into account recent changes to predict the very next changes in the system is important, as shown by the difference between association rules mining and enhanced association rules mining. The only addition to enhanced association rules mining is to also mine for rules in the current session, and the results are drastic, since its accuracy more than doubles. We tried to alter the importance of the rule based on the changes in the sessions. We found that taking into account the timing of the changes when they occured in a session decreased the accuracy at the

class level, but increases it at the method level. This may be because classes are interacted with on long portions of a development session, while method interactions are much more localized in time, and usually worked at only for brief periods of time. Hence the measure is much more useful for predicting method changes.

Degree of Interest. Considering only recent changes is sometimes not enough. We expected the degree of interest to perform better than association rules mining. Although their accuracy is comparable, association rules mining is a more accurate prediction approach. This is due both to the adaptation factor of the degree of interest when switching tasks (its initial accuracy is lower), and the fact that the association rules look in the entire past and can thus find old patterns that are still relevant. The Mylyn tool which uses a degree of interest has a built-in notion of tasks [9], that alleviate these two problems: A degree of interest is maintained for each task, and is manually recalled by the developer. Task switching recalls another degree of interest model, which may also contain information from the further past. Therefore, we need to evaluate the accuracy of several degrees of interest combined, and selecting the one best adapted to the task at hand.

Coupling. Coupling based on the system structure is overall less accurate than other approaches. This is to be expected this is does not take into account recent or past changes at all. However, it proved to be efficient when prediction approaches were combined. Using it as a second opinion significantly raised the accuracy of some approaches.

HITS. Overall, as Figure 1 illustrates, the overall best performing approach we found is the HITS algorithm, using a graph featuring the structure of the system among the recent changes (the last 50 changes). The HITS algorithm can be applied to any graph, so alternative definitions of the graph based on the same data may yield even better results. Since the graph definition we use considers both recent changes and the structure of the system, we think it is a good representative of the potential of HITS-based approaches. Nevertheless, a possible improvement would be to incorporate change data from the further past. Since HITS is, as the Degree of Interest approach, sensible to task switching, in the future we need to evaluate the accuracy of several HITS graphs combined.

5 Discussion

Despite the good results we obtained, there are a number of issues we want to discuss:

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. However, some of the results we found were in line with results found by other researchers. Hassan and Holt found that coupling-based

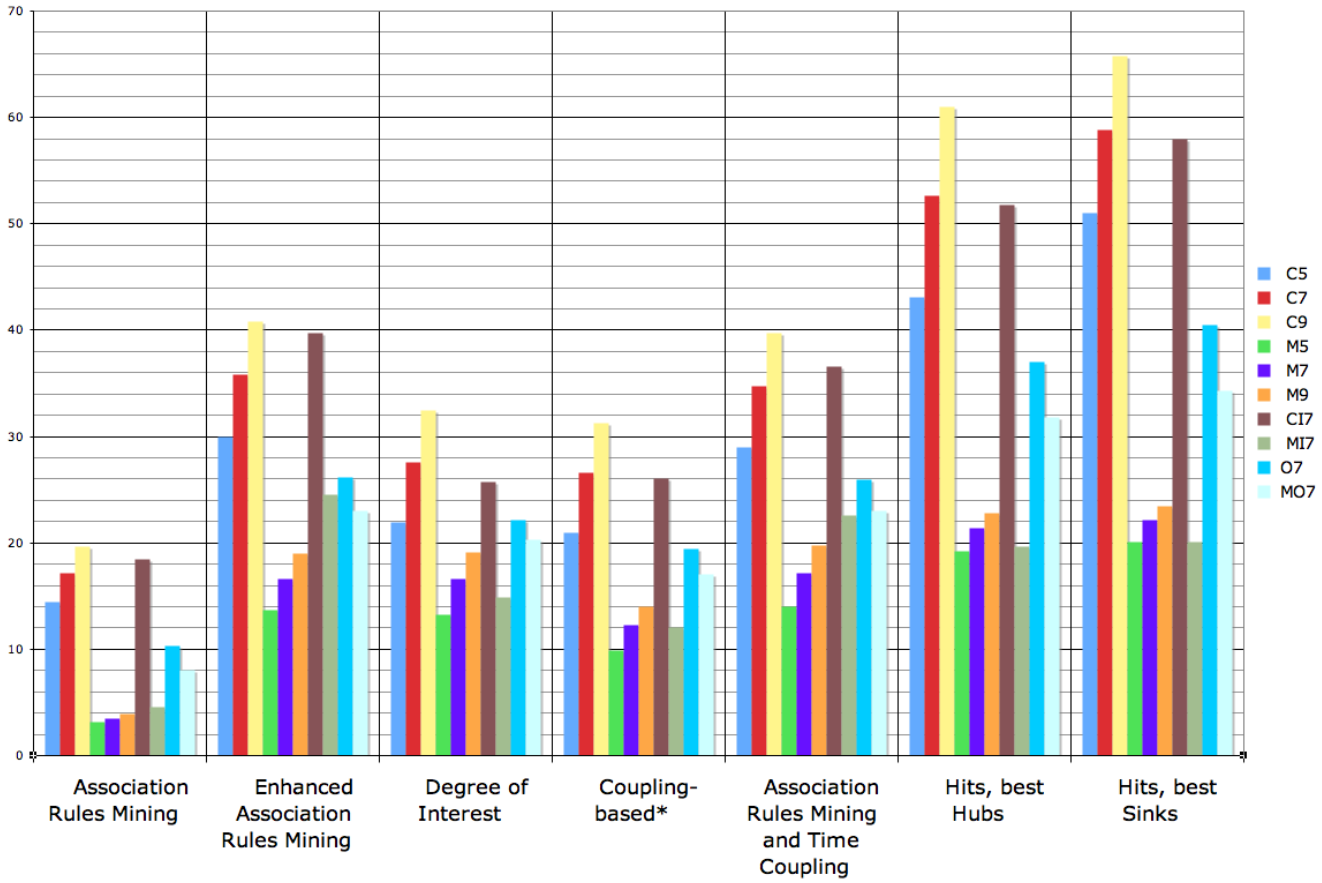

Figure 1. Prediction Results

approaches are less precise than history based approaches, and so do we. Similarly, Zimmermann *et al.* find a precision of 4% for methods during active development. Reproducing the approach with our data yields comparable results. We also found evidence supporting Kagdi's proposal of merging association rule mining and coupling-based impact analysis. We found an increase of accuracy with a simple merging strategy. A more developed merging strategy may yield better results.

Absent Data. Our data does not include navigation data, which is used in approaches such as NavTracks. Mylyn's Degree of Interest also includes navigation data. We started recording navigation data after recording changes. As such, we only have navigation data for a portion of SpyWare's history. The lack of navigation data needs to be investigated, in order to see if it impacts the accuracy of our implementation of Degree of Interest.

Evolving the benchmark. Our benchmark still needs to evolve. As said earlier, the dataset should be larger for the results to be more generalizable. Another possible enhancement to the benchmark would be to evaluate if a tool is able to predict a change's *content* and not only its location. Some possibilities would be to evaluate the size of the change, *i.e.,* whether a small or a large change is expected, or the actual content. The latter could be possible and useful to evaluate code clone management.

6 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. Our benchmark takes into account the two scenarios in which change prediction tools are used: As maintenance assistance tools to infer the locations likely to change when a location has to change, and as productivity enhancement tools when actively developing new code.

If the first case was partly covered by replaying changes from a repository, the second case was not covered at all, so far. 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 [22]. 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 changes would be an interesting variation.

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 merging strategies). We found that the HITS approach, when used on a graph comprising recent changed entities in the system and linked according to their structure, provided the most accurate predictions, with significant improvements over the state of the art.

References

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

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

[3] 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)*, pages 190–198, Los Alamitos CA, 1998. IEEE Computer Society Press.

[4] T. Gîrba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Berne, Berne, Nov. 2005.

[5] T. Gîrba, S. Ducasse, and M. Lanza. Yesterday's Weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 40–49, Los Alamitos CA, Sept. 2004. IEEE Computer Society.

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

[7] H. H. Kagdi. Improving change prediction with fine-grained source code mining. In R. E. K. Stirewalt, A. Egyed, and B. Fischer, editors, *ASE*, pages 559–562. ACM, 2007.

[8] 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)*, pages 159–168, 2005.

[9] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of SIGSOFT FSE 2006*, pages 1–11, 2006.
[10] J. Kleinberg. Authoritative sources in a hyperlinked environment. Technical report, IBM, May 1997.
[11] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Computer Science Department, Stanford University, 1998.
[12] S. Pinker. *How the Mind Works*. W. W. Norton, 1997.
[13] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Software Eng.*, 33(6):420–432, 2007.
[14] R. Robbes and M. Lanza. Characterizing and understanding development sessions. In *Proceedings of ICPC 2007 (15th International Conference on Program Comprehension)*, pages 155–164. IEEE CS Press, 2007.
[15] 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)*, pages xxx–xxx. ACM Press, 2008.
[16] 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)*, pages xxx–xxx. ACM Press, 2008.
[17] R. Robbes and M. Lanza. Spyware: A change-aware development toolset. In *Proceedings of ICSE 2008 (30th International Conference in Software Engineering)*, pages 847–850. ACM Press, 2008.
[18] R. Robbes, M. Lanza, and M. Lungu. An approach to software evolution based on semantic change. In *Proceedings of FASE 2007 (10th International Conference on Fundamental Approaches to Software Engineering)*, pages 27–41, 2007.
[19] R. Robbes, D. Pollet, and M. Lanza. Logical coupling based on fine-grained change information. In *Proceedings of WCRE 2008 (15th Working Conference on Reverse Engineering)*, pages xxx–xxx. IEEE CS Press, 2008.
[20] J. Sayyad-Shirabad, T. Lethbridge, and S. Matwin. Mining the maintenance history of a legacy software system. In *ICSM*, pages 95–104. IEEE Computer Society, 2003.
[21] Y. Sharon. Eclipseye - spying on eclipse. Bachelor's thesis, University of Lugano, June 2007.
[22] S. E. Sim, S. M. Easterbrook, and R. C. Holt. Using benchmarking to advance research: A challenge to software engineering. In *ICSE*, pages 74–83. IEEE Computer Society, 2003.
[23] 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)*, pages 325–335. IEEE Computer Society, sep 2005.
[24] N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides. Predicting the probability of change in object-oriented systems. *IEEE Trans. Software Eng.*, 31(7):601–614, 2005.
[25] G. M. Weinberg. *The Psychology of Computer Programming*. Dorset House, silver anniversary edition edition, 1998.
[26] F. G. Wilkie and B. A. Kitchenham. Coupling measures and change ripples in c++ application software. *Journal of Systems and Software*, 52(2-3):157–164, 2000.
[27] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *Transactions on Software Engineering*, 30(9):573–586, 2004.
[28] A. Zeller. The future of programming environments: Integration, synergy, and assistance. In *FOSE '07: 2007 Future of Software Engineering*, pages 316–325, Washington, DC, USA, 2007. IEEE Computer Society.
[29] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE*, pages 563–572. IEEE Computer Society, 2004.