

# Extracting Change-patterns from CVS Repositories

Salah Bouktif<sup>1</sup>, Yann-Gaël Guéhéneuc<sup>2</sup>, and Giuliano Antoniol<sup>1,3</sup>

<sup>1</sup>Département de Génie Informatique, École Polytechnique de Montréal, Canada

<sup>2</sup>P<sub>TIDEJ</sub> Team, GEODES, DIRO, University of Montreal, Canada

<sup>3</sup>RCOST, University of Sannio, Italy

salah.bouktif@polymtl.ca, guehene@iro.umontreal.ca, antoniol@ieee.org

## Abstract

*Often, the only sources of information about the evolution of software systems are the systems themselves and their histories. Version control repositories contain information on several thousand of files and on millions of changes. We propose an approach based on dynamic time warping to discover change-patterns, which, for example, describe files that change together almost all the time. We define the Synchrony change-pattern to answer the question: given a software system and one file under modification, what others files must be changed? We have applied our approach on PADL, a software system developed in Java, and on Mozilla. Interesting results are achieved even when the discovered groups of co-changing files are compared with these provided by experts.*

## 1 Introduction

Tracking changes in a software system during its evolution is a major concern because state-of-the-art software development processes, such as open source, and geographically-distributed teams make such tracking an essential but difficult activity. Indeed, teams working on a same system from different countries and with various natural languages may have difficulty in sharing timely information about the rationale of changes. Open source development is not only geographically distributed but often realised by developers working during short time intervals and thus with limited knowledge of previous changes and their rationale.

Concurrent Versions System (CVS) is a convenient and simple system to record and to coordinate changes and to manage releases and versions. CVS repositories are essential to the open-source community as well as to many companies. They contain vast amounts of data on several thousands of files and millions of

changes. This data includes the modifications on software artifacts, times of changes, developers' identities and comments, and tags to identify major and minor releases. However, they do not provide data on the rationale of changes readily.

Yet, CVS repositories offer a wealth of data to extract synthetic representations of software evolution and to build abstractions of changes. These representations and abstractions may help in answering developers' specific needs. In particular, they contain appropriate data to track changes at a higher-level of abstraction than individual changes: to track entities of the software system *changing together*. Thus, using CVS repositories, we could provide developers with hints on the rationale of previous changes and answer the important question: *If this particular file changes, what other files should change?*

This question has already been partially addressed in previous work, in particular Zimmermann's [24]. Yet, we propose a novel approach, using a technique from pattern recognition. Our approach yields to more accurate results than Zimmermann's when considering files as artifacts. We also report the first evaluation of such an approach in comparison to a *golden standard* provided by experts.

We define the general concept of change-patterns and describe one such pattern, Synchrony, that highlights *co-changing* groups of files and thus helps in answering the previous question. We restrict the definition of co-changing files to files that were modified almost at the same times, *i.e.*, around the same moments in time. The meaning of *same time* is left intentionally vague to represent time frames of minutes or hours, depending on the data in a given CVS repository. Co-changes do neither necessary imply a causal relationship among the individual changes of the different co-changing files nor logical dependencies among the files. For example, two files may change at the same time because developers decided to adopt a new li-

cense. However, it is reasonable to assume that, during the *long-term* evolution of a software system, a similar temporal pattern of changes is likely to reveal a dependency among files, hinting developers to change—or, at least, to consider—all files in a group of co-changing files when one of these is subject to change.

We use the technique of Dynamic Time Warping (DTW) [19], developed in pattern recognition and adopted in early speech recognition systems, to identify groups of co-changing files. The histories of the files of a software system in a CVS repository are pre-processed to extract windows of interest (*i.e.*, certain amounts of consecutive changes). Within these windows, we compute a distance using DTW for each pair of histories. We use this distance to filter files and to group files with similar evolution histories. We define a group of co-changing files as a group of files having a DTW distance below a given threshold. This is a simple yet accurate form of clustering, other clustering techniques will be studied as part of future work.

We perform two case studies to illustrate our approach. We realise a first case study using PADL, a Java software system with a three-year history and report results on the accuracy of the extracted Synchrony change-pattern. First, we perform an internal evaluation, in which we compare groups of co-changing files—conform to the Synchrony change-pattern—with change-patterns extracted from a testing set of data that are not involved in the grouping process. We obtain an internal precision of 84.81% and a recall of 71.86%. Second, we accomplish an external evaluation, in which we compare groups of co-changing files with these provided by PADL experts in the form of a golden standard. This external evaluation emphasises the great interest of our technique, with an average external precision of 78.98% and recall of 76.89%, for 91 PADL files. The second case study, on Mozilla, emphasises the scalability of our approach: out of 20,134 files retrieved from the CSV repository of the Mozilla project during five days starting on July 15, 2005, we retain 9,799 source files. The internal precision and recall of the groups are 73.58% and 67.61%, respectively.

The main contributions of this paper are:

- Definitions of change-patterns and of the Synchrony change-pattern for co-changing files.
- The use of dynamic time warping to identify co-changing files in CVS repositories with good precision and recall
- A comparison of the results of our approach with a golden standard provided by the developers of PADL and another case study using Mozilla.

The remainder of this paper will be organized as follows: Section 2 summarises related works. Section 3 formulates the problem of extracting change-patterns and defines the Synchrony change-pattern. Section 4 introduces the technique of dynamic time warping. Section 5 presents our approach, its process and tools. Section 6 reports the results of the case studies. Section 7 concludes and sketches future work.

## 2 Related Work

CVS repositories have already drawn the attention of several researchers. Our approach shows greater *internal* and *external* accuracies.

In [9, 10, 11], the authors used module version numbers to detect evolution patterns in CVS repositories. They used version data to also assess growth and change of behaviour [11]. They identified common change patterns across all the parts of object-oriented programs using three in-house data-mining techniques, respectively called Quantitative Analysis, Change Sequence Analysis, and Relation Analysis.

In [23], the authors further introduced the problem of co-changing files and presented an approach for their identification. They compute association rules among files by applying data mining techniques on CVS repositories. Zimmermann *et al.* [24] extended this previous work to recover co-changing fine-grain entities (classes, methods, fields. . .). They suggest likely future changes by detecting causal couplings between entities to prevent incomplete changes. German [12] abstract co-changing files into modification requests and analyses their interrelationships and authors.

In [3], the authors proposed a new approach for clustering software artifacts using historical data to produce interpretable graph layout. They defined the concept of co-change graphs and derived requirements for their layout. The results of their approach are layouts (not groups) that can be used to display co-change graphs to ease their interpretation. Similarity measures are not applicable to their results, thus they do not provide evaluation of the accuracy of their approach.

With respect to previous work, our main contributions are: (1) a definition of change-patterns in general and of the particular Synchrony change-pattern for co-changing files; (2) the use of dynamic time warping to identify co-changing files in CVS repositories; and, (3) an external evaluation of the results of our approach with a golden standard provided by the developers of PADL. The evaluation of our approach using an external golden standard is original among other works tackling the same problem. Our approach provides higher precision and recall than Zimmermann’s approach [24] when this later is applied to files.

### 3 Problem Formulation

We define change-patterns as common and recurring modifications of software systems in time, during the evolution of the systems. Change-patterns embody the relationships among changes, when considering any modification on any software artifact as a possible change. Change-patterns reify explicit and implicit dependencies among software artifacts, the logical order in which change is performed. Extracting change-patterns is important during maintenance because they provide guidance to maintainers to carry out complete and consistent modifications.

For example, let us assume that whenever a method is added to a class  $A$  of an object-oriented system, a new class is created in a package  $p$ . This information could be encoded as a change-pattern, which would help maintainers in changing consistently the system, when adding a new method to  $A$ , by highlighting the need to add a new class to  $p$ .

In the remainder of this paper, we study a particular change-pattern, which we called Synchrony, describing files that change almost at the same time. We assume that if two or more files were often changed in the past at almost the same moments in time, such co-changes are likely to happen again in the near future. Our conjecture is that dependencies, explicit and implicit, among files tend to remain stable over time.

By extracting groups of files following the Synchrony change-pattern, we can answer the question: *If this particular file changes, what other files should change?* Thus, we circumvent the need for specific knowledge on file dependencies to determine the extent of a change by identifying file histories exhibiting the Synchrony change-pattern. For example, file histories depicted in Figure 1 shows that whenever  $f_1$  changed,  $f_2$  also changed, and that whenever  $f_3$  changed, so did  $f_4$ .

The extraction of change-patterns, including the Synchrony change-pattern, from CVS repositories requires specifying a unit of time and the artifacts of interest. CVS repositories allow the extraction of fine-grain data on the changes of a software system files, directories, and their contents: up to the line of code with a precision of  $1/1,000^{th}$  of a second on Unix systems. Such fine-grain details, as a whole, is scarcely informative about the evolution of a system and can easily lead to information overflow. Therefore, we choose the second as unit of time for the granularity of changes. For the same reason, we consider files as our only artifact of interest to avoid information overflow because files are the basic units of commits in most CVS repositories.

We choose to work with files for the ease of their analyses. Yet, we could as well apply our approach on finer-grain constituents of programs, such as functions,

data structures, classes, or methods. Indeed, given the data from the CVS repository and the semantics of the programming language used to encode the files, we can reflect the changes to a file to its constituents by computing the enclosing constituents where the changes occurred, such as in Zimmermann's approach [24].

### 4 Dynamic Time Warping

Kruskal and Liberman [19] introduced the technique of *time warping* to compute distances between two curves. Time warping allows matching two curves subject to alterations and to variations in speed from one portion to another. It can warp the time axis, compressing time in some intervals and expanding time in others. The technique of dynamic time warping uses dynamic programming to compute time warping on signals, *i.e.*, non-linear mapping of one signal onto another, minimising the distance between the two signals.

Let  $S_1$  and  $S_2$  be two signals described by the two time series:

$$S_1(n), n = 1, 2..N, S_2(m), m = 1, 2..M.$$

We can construct a warp path  $W$

$$W = w_1, w_2, \dots, w_P$$

where  $P$  is the length of the warp path ( $\max(N, M) \leq P < N + M$ ). The  $k^{th}$  element of the warped path is  $w_k = (i, j)$ , where  $i$  is an index in the time series  $S_1$  and  $j$  in  $S_2$ . The warp path must start at the beginning of each time series at  $w_1 = (1, 1)$  and end at  $w_K = (N, M)$ . The optimal warp path is the path for which the following distance reaches its minimum:

$$Dist(W) = \sum_{k=1}^{k=P} Dist(w_{ki}, w_{kj})$$

where  $Dist(W)$  is the overall distance between the two time series  $S_1$  and  $S_2$ , the optimal distance along of warp path, and  $Dist(w_{ki}, w_{kj})$  is the distance between two elements of  $S_1$  and of  $S_2$  for the  $k^{th}$  element of the warp path. In the rest of this work, for simplicity, we use the euclidian distance.

Dynamic programming and dynamic time warping have been applied successfully to a number of problems, including optimal control [2, 7], speech recognition [16, 21], medicine [4], robotics [20], handwriting recognition [22], time-series indexing, and data mining [17, 18]. Recently, DTW has been applied to vertical profiling [15], the process of analysing and of understanding performances of modern multi-layer programs, where the authors uses DTW to align traces obtained from separate runs and correlate (among other correlations) performances data.

## 5 Change-patterns Extraction

Our approach to extract the Synchrony change-pattern in order to identify co-changing files consists of three steps. First, we gather relevant data from a CVS repository. Second, we transform the data before its use in the third step. In particular, we compute an appropriate window length *to limit* the effect of past changes. Indeed, we group files together according to a measure of the distances of their histories. Long histories are likely to contain changes that happened too long ago and that do not provide pertinent information on current dependencies. Finally, using the DTW technique, we identify groups of co-changing files conforming to the Synchrony change-pattern. We rely on the tool chain from our previous study [1]. The following subsections detail the three steps of our approach.

### 5.1 Gathering Relevant Data

Mining a CVS repository is performed using `cvs2cl.pl`<sup>1</sup>, a Perl script dedicated to extracting data from CVS repositories. First, a remote CVS repository is mirrored locally using appropriate CVS commands. Second, for each file in the mirrored repository, a detailed change log encoded in XML is produced.

Change logs extracted from the CVS repository are post-processed by means of a Java program to produce file histories. Each file history is a list of the moments in time in which changes in the file were committed, from the most recent to the oldest change.

Time histories are analysed and each date of change is mapped in the number of seconds from the *epoch*, the conventional zero in Unix systems (Jan 1, 1970).

### 5.2 Computing Appropriate Windows

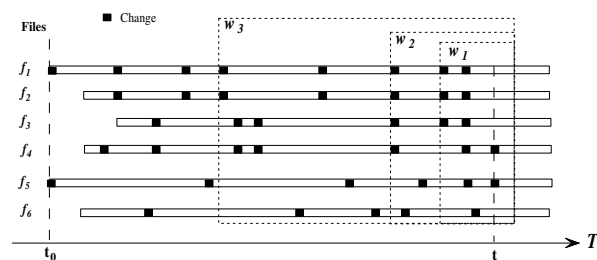


Figure 1. File change history.

The number of changes that files underwent is a variable spread fairly evenly: Certain files are never changed while other are constantly modified. Furthermore, as software systems evolve, dependencies are likely to change, *i.e.*, more recent changes are likely to

reflect a more recent structure of dependencies. Yet, changes happened in the past may be representative of some deeper dependency, even if relatively old because the related files reached a certain stability.

As shown in Figure 1, two files,  $f_1$  and  $f_2$ , have a pattern of co-changes dating back to  $t_0$ . When the window  $w_3$  is applied only  $f_1$  and  $f_2$  are *really* co-changing. However, if less changes are considered, as with window  $w_2$ , a co-change can be noticed among files  $f_1$ ,  $f_2$ , and  $f_3$ . Thus, a compromise is required between keeping track of all past changes but missing some more recent co-changes and studying only recent co-changes but forgetting past dependencies. In Figure 1, we must compromise between using window  $w_3$ , highlighting the co-changes of  $f_1$  and  $f_2$ , possibly a now-obsolete dependency, and window  $w_2$ , circumscribing co-changing files  $f_1$ ,  $f_2$ , and  $f_3$ .

The choice of an appropriate window depends on the distributions, in time, of changes, for each file. It can be performed essentially in two ways: either by imposing a fixed distance in time (*i.e.*, changes not older than one year) or by selecting a maximum fixed number of consecutive past changes. A window of fixed number of changes is easily implemented and ensures that both recent and past changes are retained.

In our experiments, we observe that the difference between two consecutive changes, although subject to some randomness, tends to be low, *i.e.*, most files are changed frequently. When two far-apart changes are considered, *e.g.*, when four or five changes happened in between, the distributions of the changes in time tend to be more spread, as expected. For each file history, we analyse the cumulative distributions of  $Change[m] - Change[m - (r + 1)]$  where  $m$  is a moment in a file history and  $r \in [1..5]$ . In Figure 3, for the system in our first case study, we show that, for values of  $r$  up to five, the corresponding windows of five to seven changes capture all recent changes but also retains a sizeable fraction of more recent changes.

### 5.3 Grouping of Co-changing Files

The grouping of co-changing files is realised using the DTW technique. This technique is used to align each possible pair of file histories, expressed in seconds, and to group together histories having a *distance* below a given threshold. A DTW algorithm does not necessarily provide a distance, *i.e.*, distance axioms may not be valid depending on the implementation. Our implementation ensures that  $Dist(S, S) = 0$  and  $Dist(S_1, S_2) = Dist(S_2, S_1)$ . The cost of the DTW technique is quadratic in the length of the file histories. Yet, due to symmetries, only half of all the comparisons between file histories is required.

<sup>1</sup><http://www.red-bean.com/cvs2cl/>

```

threshold ← value
for each hist1 = f1, f2, ... do
  Sethist1 ← ∅
  for each hist2 = f1, f2, ... do
    dist ← DTW(hist1, hist2)
    if (dist < threshold) then
      Sethist1 ← Sethist1 ∪ hist2
  endfor
endfor

```

**Figure 2. DTW grouping pseudo-code.**

Figure 2 shows the pseudo-code of our DTW algorithm. This pseudo-code is similar to fuzzy clustering, where an element can belong, with different strengths, to several clusters. More work is needed to analyse the strengths of the groups, we will devote future work to study alternative clustering and grouping techniques. In our current use of the DTW technique, we do not force a file to participate in one and exactly one group because we have no a-priori evidence that co-changes are symmetric and transitive.

Beside the number of changes to be retained, *i.e.*, the size of the window, the only parameter of the DTW algorithm is the cutting threshold to assign a file to a group of co-changing files. Preliminary experiment suggested that a value between 43,200 seconds (12 hours) and 172,800 seconds (48 hours) would provide accurate results. Fine-grain analysis performed on PADL shows that a compromise between precision and recall is needed because lower values for the threshold tend to privilege precision. Values between 43,200 seconds and 86,400 seconds (24 hours) are a good compromise, slightly privileging recall. A threshold above 86,400 seconds privileges recall over precision.

The DTW algorithm was implemented in C. We deem computation times acceptable for our case studies on PADL and Mozilla. All file comparisons on PADL require few seconds only. The grouping of 9,799 files from Mozilla is performed in less than 180 seconds (3 minutes) on a Pentium laptop running RedHat.

## 6 Case Study

We analyse the history of PADL, a software system developed in Java, to evaluate the internal precision and recall of the results of our approach. We also compare the precision and recall of the results with a golden standard. To the best of our knowledge, this is the first comparison with a golden standard of the results of an approach to analyse software evolution reported in the literature. In addition, we include a study of the scalability of our approach using Mozilla.

### 6.1 Subject of the Case Study

PADL (*Pattern and Abstract-level Description Language*) [13] is a meta-model providing constituents to describe object-oriented programs and the structures of the solutions of design patterns.

The meta-model appeared in 1999 under the code-name PDL (*Pattern Description Language*) as a small set of Java classes developed by one programmer, Hervé Albin-Amiot, to instantiate the structures of the solutions of design patterns. It evolved to become a complete meta-model to describe class diagrams in 2001 with the work of Albin-Amiot and Guéhéneuc. In 2002, it changed to a meta-model to describe program models (in addition to the structures of solutions of design patterns) at different levels of abstraction with the addition of a parser for Java and of precise definitions binary class relationships [14]. Since 2003, it has been under constant development under the direction of Guéhéneuc, with the contributions of about 10 people and is used in many research projects at University of Montreal and elsewhere.

The first releases of PDL were performed in an *ad hoc* manner, without the use of any system of source configuration management. Since 2003, PADL is stored in a CVS repository, hosted at University of Montreal with a public access<sup>2</sup>, in the form of several modules corresponding to Eclipse projects. Of particular interest for the experiments is module PADL, storing the main PADL project, which contains the core packages and classes of the meta-model.

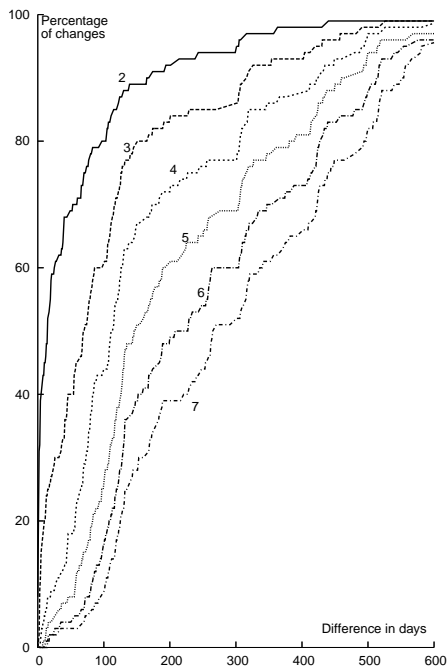
PADL size is 14 KLOC, it contains 136 files organised in 23 repositories. 91 files, with five or more changes, are retained for the experiment.

### 6.2 Objective of the Case Study

The objective of the case study is four-fold: (1) to compute the internal precision and recall of the results of our approach; (2) to compare the results with a golden standard and to compute their external precision and recall; (3) to assess the predictive power of our approach; and, (4) to study the scalability of our approach on an industrial-size program. Hence, we planned and performed the three following activities:

1. A  $k$ -fold cross validation of the results.
2. Comparison of the results with a golden standard.
3. Comparison of the prediction of changes with the golden standard.

<sup>2</sup>Please visit [cvsiro.iro.umontreal.ca/cgi-bin/cvsweb.cgi](http://cvsiro.iro.umontreal.ca/cgi-bin/cvsweb.cgi) to browse the public CVS repository and [ptidej.iro.umontreal.ca/forparticipants/development/cvseclipse](http://ptidej.iro.umontreal.ca/forparticipants/development/cvseclipse) for instructions on accessing the repository.



**Figure 3. Distribution of time differences in PADL for windows of various lengths.** A number  $r$  indicates the distances between *two* changes, hence corresponds to a window of  $r + 2$ .

4. A  $k$ -fold cross validation of the results of our approach on a large software system with a long history, the Mozilla Web browser.

The results of our approach and of their comparison with the golden standard have been assessed using two widely accepted information retrieval measures, *precision* and *recall* [8]. *Recall* is the ratio of the number of relevant documents retrieved for a given query over the total number of relevant documents for that query. *Precision* is the ratio of the number of relevant documents retrieved over the total number of documents retrieved. In our case study, in which we identify groups of co-changing files, the query corresponds to the file to be changed and the set of retrieved documents corresponds to the groups of co-changing files.

Several queries are possible; average precision and recall were computed with the arithmetic mean of the values (leading values  $Prec_a$  and  $Rec_a$ ) and of the weighted values, computed as follows:

$$Prec_w = \frac{1}{\sum_q |Group(q)|} \times \sum_q |Group(q)| \times Prec(q)$$

$$Rec_w = \frac{1}{\sum_q |Group(q)|} \times \sum_q |Group(q)| \times Rec(q)$$

where  $Prec(q)$  and  $Rec(q)$  are the precision and recall of the query  $q$ ,  $|Group(q)|$  the size of the  $q^{th}$  group.

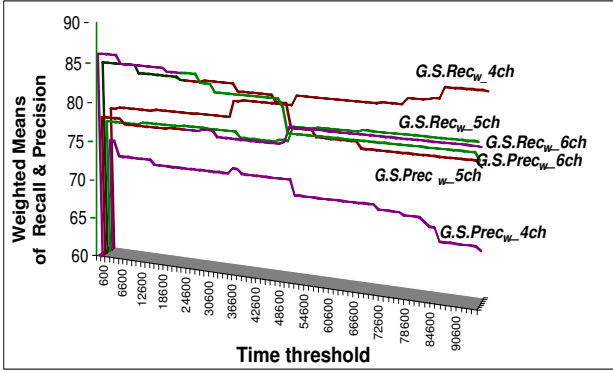
### 6.3 Experimental Setting

We observed that histories with too few changes, *e.g.*, a single change, would lead to non meaningful results because a single change does not represent a history. To the contrary, histories with a large number of changes in the far past may not be relevant with respect to recent changes. Thus, we choose histories of lengths greater or equal to five change and shorter or equal to seven changes. Histories longer than seven changes were truncated at seven changes.

To build and to validate the results of the grouping, training and testing sets are obtained by defining different cutting points for the histories of lengths five, six, and seven. We identify each configuration of a testing set by a triplet  $a - b - c$ , where  $a$ ,  $b$ , and  $c$  are the lengths in number of changes of the sub-history constituting the testing set for histories of length five, six, and seven, respectively. For example, the triplet 3-2-4 corresponds to groups obtained using the three most recent changes of any history of length five as a testing set and, similarly, to groups using the two and four most recent changes of histories of lengths six and seven, respectively.

With histories of length seven and testing sets of length five, we would have only two changes—the oldest changes—in the training set. We believe that these two changes, if used to compute the five subsequent changes, cannot produce accurate results because they are too far in the past to be relevant to recent changes, thus we impose  $c < 5$ . For similar reasons, we impose that  $a > 1$ ,  $b > 1$  and  $c > 1$ . Then, we enumerate exhaustively each triplets satisfying these constraints and use these triplets in this case study.

We perform a study of the influence of the threshold on precision and recall to verify our preliminary findings concretely. We mimic a real world application by taking the first seven changes of the history of PADL, the last change is ignored to produce histories up to six changes. This setting corresponds to predicting the next change from past changes. Precision and recall were compared with the golden standard. We vary the threshold between 600 and 96,000 seconds with steps of 1,200 seconds (20 minutes). Figure 4 shows the precision and recall for the variations of the thresholds. Low values tend to privilege precision while higher threshold values lead to higher recall. We choose a threshold of 86,400 seconds, the extreme right of the flat part with constant precision and recall. This is a compromise that privileges recall over precision slightly.



**Figure 4. Variations of the precision and recall with different thresholds.**  $G.S.Prec_w-\mathcal{X}ch$  ( $G.S.Rec_w-\mathcal{X}ch$ , respectively) are the weighted means for precision (recall) using the golden standard, with a window of length  $\mathcal{X}$ .

## 6.4 Activity 1 – Internal Evaluation

The rationale behind the internal evaluation of our approach is the hypothesis that no expert and no pre-existing groups of co-changing files are available. A particular case of the  $k$ -fold cross validation method is used to assess accurately the precision and recall of the results of our approach. In this cross-validation, training—the oldest part of the history—and test—the latest part of the history—are kept distinct and no overlapping change exists. We build groups from the training sets (*e.g.*, the four oldest changes in a history of seven) and from the testing sets (*e.g.*, the latest three changes in a history of seven). Precision and recall are measured for the groups resulting from the training sets by considering, for each file, the groups resulting from the testing sets as oracles, *i.e.*, the *correct* groups of co-changing files.

For each of the retained configurations, we compute the weighted and the arithmetic means of the precisions ( $Prec_w$  and  $Prec_a$ , respectively) and of the recalls ( $Rec_w$  and  $Rec_a$ , respectively) of the trained groups. We report in Table 1, averages and standard deviations of  $Prec_w$ ,  $Prec_a$ ,  $Rec_w$ , and  $Rec_a$ . The upper part of Table 1 shows that our approach provides interesting results with a minimum average precision of 84.81% and recall of 71.86% for  $Prec_w$  and  $Rec_w$ . These results should be compared with Zimmermann’s results (when considering files as artifacts) that have a precision of 26.00% and a recall of 30.00%.

## 6.5 Activity 2 – External Evaluation

We believe that the evaluation of the accuracy of approaches such as ours should be performed by comparing their results with golden standards because only experts in the software system under study and in its development process can assess the quality of the groups. This is a manual and time-consuming task that was performed on PADL. We asked the second author, an expert-developer of the PADL meta-model, to define ideal groups as golden standard. We took pain to ensure that the expert remains unbiased towards the results obtained with our approach. Hence, the expert did participate in no way in the computation of the groups. He also did not have access to the details provided here before the analysis of the groups.

The golden standard is provided by the expert in the form of groups of co-changing files. The expert identified files that co-changed in the past and that were likely to co-change in the near future to build this golden standard. This identification raised the problem of the semantics of changes. Indeed, it is difficult for an expert to assess the files that *should* be changed along a given file without any knowledge of the *changes* performed. This problem is similar to the problems of change impact analysis [5] and of refactoring detection [6]. It is made even more acute by the consideration of *time*. Indeed, the files changing along a key file may vary depending on the moment in time of the change (and on the particular change). The following group from PADL illustrates this problem:

```

1      ./src/padl/kernel/impl/MemberClass.java
2      -> ./src/padl/kernel/impl/MemberClass.java
3      ./src/padl/kernel/impl/MemberInterface.java
4      ./src/padl/kernel/impl/ContainerAggregation.java

```

In this group, the expert expresses that each time the file `MemberClass.java` was changed, the files `MemberInterface.java` and `ContainerAggregation.java` were changed as well. This group describes a logical dependency because the concepts of member entity (class and interface) have been recently introduced in the meta-model and required changes to the implementation of the concept of `ContainerAggregation`.

However, from this moment in time on, the (pseudo-)causal relationship among the three files may no longer be valid. The relationship would rather be, for example, between `MemberClass.java` and `IMemberClass.java` (its interface of definition). Yet, the relationships depends on the extent of the change performed: A major change to the file `MemberClass.java` could impact its interface `IMemberClass.java` as well as other related files, such as `MemberInterface.java`.

Thus, the expert kept a very conservative approach when building the groups for the golden standard. He attempted to list all groups of co-changing files by enumerating combinatorially all recent files and using his knowledge of PADL and of recent changes. Then, these groups and the groups built during the  $k$ -fold cross validation are used to compute precision and recall, with reference to the golden standard. We perform the same computations as for the internal evaluation and report, in the lower part of Table 1, averages and standard deviations in comparison to the golden standard.

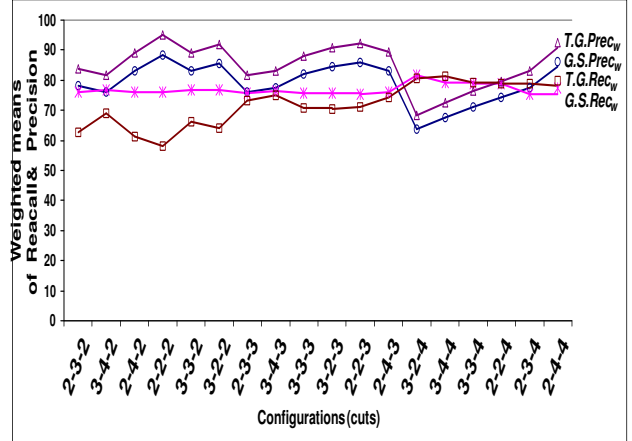
The results in Table 1 show relatively high precision and recall despite the difficulty of the grouping problem. By comparing the results of the internal evaluation with those of the external evaluation, we notice that the two approaches of grouping give very close and high accuracies. In particular, differences between respective averages of precision and recall are always smaller than one standard deviation. Moreover, the curves  $G.S.Prec_w$  and  $T.G.Prec_w$ , in Figure 5, of the precisions computed on the groups in the golden standard and those from the testing sets have similar shapes. The same similarity is also observed for the curves of the recalls, but slightly less clearly. Thus, we conclude that for larger-scale software systems, our approach may yield accurate groups of co-changing files.

	$Prec_w$	$Prec_a$	$Rec_w$	$Rec_a$
Internal Evaluation	84.81 (7.02)	85.91 (2.47)	71.86 (6.98)	59.32 (0.26)
External Evaluation	78.98 (6.52)	91.10 (2.54)	76.89 (1.66)	64.77 (7.03)

**Table 1. Experimental results for PADL. The mean (and standard deviation) of the precisions and recalls from the internal and external evaluations.**

## 6.6 Activity 3 – Predictive Power

An on-line system to predict co-changing files would have to provide an estimate of the files to be changed given the past histories of the files and a file being changed. To mimic such a situation we compute groups on PADL as when assessing the influence of the DTW threshold. We ignore from the histories of the files the last observed change. We use this shortened histories of changes to build groups with different lengths of windows, starting with a length of two changes. Figure 6 shows the trade-off between the precisions and recalls when compared to the golden standard.



**Figure 5. Variation of recalls and precisions using  $k$ -fold internal validation and external comparison with the golden standard.**  $G.S.Prec_w$  ( $G.S.Rec_w$ , respectively) are the weighted means for the precisions (recalls, respectively) using the golden standard while  $T.G.Prec_w$  ( $T.G.Rec_w$ , respectively) are the means of groups using the testing sets.

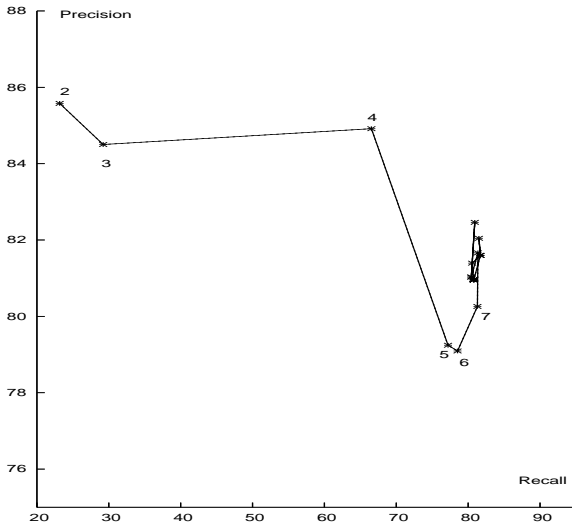
Windows	$Prec_w$	$Rec_w$
4	66.54	84.92
5	77.20	79.20
6	78.00	78.00

**Table 2. Experimental results for PADL. Weighted precision and recall of predictions.**

Different windows lengths lead to difference accuracies. Indeed, shorter windows promote recall. When precision is important a window of more than four or five changes should be adopted. Table 2 reports the weighted averages of the precisions and recalls on PADL, when predicting co-changing files. The sudden drop in precision between windows of four and five changes needs further investigations. Five changes correspond to going back in time between 200 and 350 days depending of the file; at that time PADL underwent a major restructuring. A similar abrupt variation is also observed in Figure 3 in which cumulative distributions of time differences have a sharp increase around 300 days.

## 6.7 Activity 4 – Scalability with Mozilla

We apply our approach to Mozilla to validate our approach on a large-scale software system. Mozilla is an



**Figure 6. Trade off between precision and recall for predictions.**

open-source Web browser ported on almost all known software and hardware platforms. It is sufficiently large and mature to represent a real-world program. Its size ranges in the millions of lines of code (MLOC). It is developed mostly in C++. C code accounts for a small fraction of the program. We include in our study C++, C, header files, Java and IDL, HTML, and configuration language and support. The CVS repository of Mozilla was mirrored on July 15, 2005 and histories of files extracted over the subsequent five days. On July 15, Mozilla repositories was made up of 20,134 files organised into 2,480 subdirectories.

$Prec_w$	$Prec_a$	$Rec_w$	$Rec_a$
73.58	82.74	67.61	52.18
(14.52)	(7.89)	(9.14)	(16.14)

**Table 3. Experimental results for Mozilla. The mean (and standard deviation) of the precisions and recalls from internal evaluation.**

Although the results (precision and recall) reported in Table 3 for Mozilla are slightly lower than those for PADL, they are considered as good, given the size of Mozilla and the number of changes undergone.

## 6.8 Threats to Validity

Despite the good results obtained when applying our approach to PADL, we cannot claim that our approach will give similar results for any software system.

We need to address external validity by extending our initial case study to many categories of software systems such as systems of different sizes, from different programming languages, with commercial and open-source components. The application of our approach to Mozilla is a precious step towards external validity, yet only the internal evaluation was possible with such large-scale software system at the time of this study.

While PADL has a small size and is far to be representative of all the software categories, its advantage resides in the availability of expert-developers that help us in two principal activities in this work. In addition to providing a golden standard, which is a time-consuming task, they help by highlighting particular moments in the evolution of PADL, specifically the moment in time when PADL underwent a major refactoring. Without this kind of information, we could have been surprised by our findings. Being aware of these changes allow judging better the effect of imprecise grouping of co-changing files, resulting from large-merge commitments, *i.e.*, evolution singularities. Thus, we believe that to improve results, a preprocessing activity is necessary to get information on atypical moments in the evolution of a system.

## 7 Conclusion

We defined change-patterns as common and recurring modifications of software systems in time. We have presented an approach to extract the Synchrony change-pattern, which describes co-changing files. Groups of files conforming to the Synchrony change-pattern, *i.e.*, co-changing files, are an answer to the frequent and important question: *If this particular file changes, what other files should change?*

Our approach relies on the technique of dynamic time warping to group files with histories of changes of different lengths. We obtained precisions and recalls that are higher than previously published results, for both an internal evaluation and, for the first time, for an external evaluation by experts. Indeed, we obtained an external precision of 78.98% and a recall of 76.89%. For the prediction of files to co-change, we achieved an external precision of 77.20% and recall of 79.20%, with windows of average size of 5 when histories of up to five changes were used to build the groups.

Although quadratic in nature, our approach is scalable. Groups of co-changing files were extracted from the histories of 9,799 files from Mozilla in less than three minutes, with an internal precision and recall of 73.58% and 67.61%, respectively. Yet, our approach has an average precision and recall that could be significantly improved by using clustering techniques in addition to DTW.

Future work includes the evaluation of the proposed approach on other software systems and the in-depth study of the observed fluctuations in precision and recall. It also includes building golden standards for other systems, studying the strengths of the groups, using clustering techniques, and comparing time intervals with numbers of changes for windowing.

## Acknowledgments

Giuliano Antoniol and Salah Bouktif were partially supported by NSERC, Canada Research Chair in Software Change and Evolution. Yann-Gaël Guéhéneuc was partially supported by NSERC, Discovery Grant.

## References

- [1] G. Antoniol, F. Rollo, and G. Venturi. Detecting groups of co-changing files in cvs repositories. In *International Workshop on Principles of Software Evolution*, pages 23–32, Lisbona, Portugal, Sept 2005.
- [2] R. E. Bellman and S. E. Dreyfus. *Applied Dynamic Programming*, volume 1. Princeton University Press, Princeton NJ, 1962.
- [3] D. Beyer and A. Noack. Clustering software artifacts based on frequent common changes. In *IWPC*, pages 259–268. IEEE Computer Society, 2005.
- [4] E. Caiani, A. Porta, G. Baselli, M. Turiel, S. Muzupappa, F. Pieruzzi, C. Crema, A. Malliani, and S. Cerutti. Warped-average template technique to track on a cycle-by-cycle basis the cardiac filling phases on left ventricular volume. *IEEE Computers in Cardiology*, 1998.
- [5] M. A. Chaumon, H. Kabaili, R. K. Keller, F. Lustman, and G. Saint-Denis. Design properties and object-oriented software changeability. In J. Ebert and C. Verhoef, editors, *proceedings of the 4<sup>th</sup> Conference on Software Maintenance and Reengineering*, pages 45–54. IEEE Computer Society Press, February 2000.
- [6] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In D. Lea, editor, *proceedings of 15<sup>th</sup> conference on Object-Oriented Programming Systems, Languages and Applications*, pages 166–177. ACM Press, October 2000.
- [7] S. Dreyfus and A. Levy. *The art and theory of dynamic programming*. Academic Press, New York, 1977.
- [8] W. B. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, Englewood Cliffs, NJ, 1992.
- [9] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 190, Washington, DC, USA, 1998. IEEE Computer Society.
- [10] H. Gall, M. Jazayeri, R. Klösch, and G. Trausmuth. Software evolution observations based on product release history. In *ICSM*, pages 160–, 1997.
- [11] H. Gall, M. Jazayeri, and J. Krajewski. Cvs release history data for detecting logical couplings. In *IW-PSE*, pages 13–23, Washington, DC, USA, 2003. IEEE Computer Society.
- [12] D. M. German. An empirical study of fine-grained software modifications. *Journal of Empirical Software Engineering*, 2005.
- [13] Y.-G. Guéhéneuc. Ptidej: Promoting patterns with patterns. In *proceedings of the 1<sup>st</sup> ECOOP workshop on Building a System using Patterns*. Springer-Verlag, July 2005.
- [14] Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In D. C. Schmidt, editor, *proceedings of the 19<sup>th</sup> conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, October 2004.
- [15] M. Hauswirth, A. Diwan, P. F. Sweeney, and M. C. Mozer. Automating vertical profiling. In R. P. Gabriel, editor, *proceedings of the 20<sup>th</sup> conference on Object-Oriented Programming Systems Languages and Applications*, pages 281–296. ACM Press, October 2005.
- [16] F. Itakura. Minimum prediction residual principle applied to speech recognition. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 23(1):67–72, 1975.
- [17] E. Keogh. Exact indexing of dynamic time warping, 2002.
- [18] E. J. Keogh and M. J. Pazzani. Scaling up dynamic time warping for data-mining applications. In *Knowledge Discovery and Data Mining*, pages 285–289, 2000.
- [19] J. B. Kruskal and M. Liberman. The symmetric time-warping problem: from continuous to discrete. In D. Sankoff and J. B. Kruskal, editors, *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, 1983.
- [20] T. Oates, M. D. Schmill, and P. R. Cohen. A method for clustering the experiences of a mobile robot that accords with human judgments. In *Proceedings 17th National Conference on Artificial Intelligence*, pages 846–851. AAAI Press, 2000.
- [21] L. Rabiner, E. Rosemberg, and S. Levinson. Consideration in dynamic time warping for discrete word recognition. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 26(6):575–582, 1978.
- [22] T. M. Rath and R. Manmatha. Word image matching using dynamic time warping. In *Proceedings of the Conference on Computer Vision and Pattern Recognition 2003*, volume 2, pages 521–527, 2003.
- [23] A. T. T. Ying. Predicting source code changes by mining revision history. Master’s thesis, University of British Columbia, Canada, October 2003.
- [24] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE*, pages 563–572, 2004.