Detecting Asynchrony and Dephase Change Patterns by Mining Software Repositories

Fehmi Jaafar\textsuperscript{a,b}, Yann-Gaël Guéhéneuc\textsuperscript{a}, Sylvie Hamel\textsuperscript{b}, and Giuliano Antoniol\textsuperscript{c}

\textsuperscript{a} PTIDEJ Team, DGIGL, École Polytechnique de Montréal, Québec, Canada
\textsuperscript{b} LBIT Lab., DIRO, Université de Montréal, Québec, Canada
\textsuperscript{c} SOCCER Lab., DGIGL, École Polytechnique de Montréal, Québec, Canada

SUMMARY

Software maintenance accounts for the largest part of the costs of any program. During maintenance activities, developers implement changes (sometimes simultaneously) on artefacts in order to fix bugs and to implement new requirements. To reduce this part of the costs, previous work proposed approaches to identify the artefacts of programs that change together. These approaches analyse historical data, mined from version control systems, and report change patterns, which lead at the causes, consequences, and actors of the changes to source code files. They also introduce so-called change patterns that describe some typical change dependencies among files. In this paper, we introduce two novel change patterns: the Asynchrony change pattern, corresponding to \textit{macro co-changes (MC)}, \textit{i.e.}, of files that co-change within a large time interval (change periods), and the Dephase change pattern, corresponding to \textit{dephase macro co-changes (DC)}, \textit{i.e.}, macro co-changes that always happen with the same shifts in time. We present our approach, that we named Macocha, to identify these two change patterns in large programs. We use the k-nearest neighbor algorithm to group changes into change periods. We also use the Hamming distance to detect approximate occurrences of Macro co-changes and Dephase macro co-changes. We apply Macocha and compare its performance in terms of precision and recall with UMLDiff (file stability) and Association Rules (co-changing files) on seven systems: ArgoUML, FreeBSD, JFreeChart, Openener, SIP, XalanC, and XercesC, developed with three different languages (C, C++, and Java). These systems have a size ranging from 532 to 1,693 files and during the study period they have undergone 1,555 to 23,944 change commits. We use external information and static analysis to validate (approximate) Macro co-changes and Dephase macro co-changes found by Macocha. Through our case study, we show the existence and usefulness of these novel change patterns to ease software maintenance and, potentially, reduce related costs.

KEY WORDS: Change Pattern; Co-changes; Stability; Change Period; Bit Vectors.

\textsuperscript{1}Correspondence to: Fehmi Jaafar – Jaafarfe@iro.umontreal.ca
1. Introduction

Any program must change to meet new requirements and user needs. Developers must continually adapt programs else they become progressively unsuitable [LB85] and eventually become obsolete to the point of disappearing.

Previous literature describes several approaches to extract and analyse dependencies among artefacts and to infer the patterns that describe their changes [BKZ10, KR11]. An artefact is the result of any activity in the software life-cycle used or produced by a software development process such as requirements, architecture model, design specifications, source code files, classes in object-oriented programs, and test scripts. As in previous work, e.g., [ZWDZ04], [ARV05], and [SBA06], for the sake of simplicity, we focus on C, C++, and Java source code files (.c, .cpp, and .java) as they are among the most common and popular programming languages.

Several of the previous approaches identify co-changes among files, e.g., [YMNCC04], [ZWDZ04], and [SBA06], which represent the dependencies between files that have been observed to frequently change together. Two files are co-changing if they were changed by the same developer and with the same log message in a time-window of less than 200 milliseconds [ZWDZ04]. Mockus et al. [MFH02] defined the proximity in time of changes by the check-in time of files that differ by less than three minutes. Other studies (e.g., [FPG03] and [Ger06]) described issues about identifying atomic change sets and supposed that they differed by a few minutes. Change patterns are motifs that highlight co-changing groups of files [SBA06] and that describe the (often implicit) dependencies or logical couplings among files that have been observed to frequently change together [GHJ98].

In our previous paper [JGHA11], we showed that previous approaches could not detect change patterns between files in long time intervals and–or performed by different developers and with different log messages, these approaches miss a variety of co-changed files. Yet, we also showed that such new change patterns provide interesting information to developers. For example, in the Bugzilla of ArgoUML, the bug ID 5378 states, in relation to ArgoDiagram.java, that an “ArgoDiagram should provide constructor arguments for the concrete classes to create”, which relates to ModeCreateAssociationClass.java. The bug report confirms that these two files have a relationship, which is hidden because we could not detect dependencies between these two files by static analysis. However, no previous approach can detect that these files co-changed because they were maintained by the same developer, but their changes were always separated by a few hours. Knowing the dependency among these files is useful to a new developer that must change ArgoDiagram.java: she must, also, assess ModeCreateAssociationClass.java for change. Indeed, Vanya et al. [VPV11] found that, depending on the commit practices used, a suitable time intervals between check-in timestamps of files has to be determined and leveraged to reliably approximate change sets.

Let F1 and F2 be two files of the same program, the scenario illustrated with ArgoUML could happen when a developer is in charge of a subset of a large program composed of, among others, files F1 and F2. She may change and commit these two files in the same day but with a few hours between each commit, as illustrated in Figure 1. This scenario may repeat for years and would be
undetected by previous approaches, which use sliding windows of a few minutes to group changes committed by the same developer and with the same log message. Yet, such co-change situations contains important information both for the developer and her colleagues: to avoid introducing bugs into programs, changes to F1 must likely propagate to F2, because these two files have a, possibly hidden, dependency.

As another example in ArgoUML, we found that the developers D1 and D2 contributed with some patches that contain NotationUtilityJava.java and ModelElementNameNotationUml.java\(^4\) and the bug ID 2926\(^5\) confirms that the two files have dependencies (see Section 4 for

\(^4\)http://argouml.tigris.org/issues/showattachment.cgi/2118/20101116-patch-notation.txt
\(^5\)http://argouml.tigris.org/issues/show_bug.cgi?id=2926
details). No previous approach can detect that these files co-changed because, during the development and the maintenance of ArgoUML, these two files were never changed by the same developer at the same time but were always changed by developers D1 and D2 in two consecutive times: first NotationUtilityJava.java and, subsequently, after some hours, ModelElementName-NotationUml.java, pointing out dependency among these files.

This previous scenario from ArgoUML happens when a developer D2 is reminded to change file F2 to correct a bug after some time by developer D1, whenever D1 changed file F1, as illustrated in Figure 2. Previous work, e.g., [YMNCC04], [ZWDZ04], [CCCDP10a], does not consider co-changed files if they were changed by two different authors in the same period even though knowing the, possibly hidden, dependency among such co-changed files could prevent developers from releasing a program with a bug because of a mismatch between files F1 and F2.

In this paper, we describe Macocha, an approach for detecting two novel change patterns, summarised in our previous work [JGHA11] and detailed in Section 2. Macocha, inspired from a previous work [SBA06], defines and detects the Asynchrony change pattern, macro co-changes (MC), and the Dephase change pattern, dephase macro co-changes (DC). It builds on previous work on co-changes and uses the concept of change periods, detailed in Section 2, and defined as a set of changes committed by developers in a continuous period of time. In particular, we use the k-nearest neighbor, \( KNN \), algorithm [Das91] to group changes into their change periods.

The Asynchrony change pattern (MC) describes a set of files that always change together in the same change periods. The Dephase change pattern (DC) describes a set of files that always change together with some shift in time in their periods of changes. We also consider approximate MC and DC using the Hamming distance. Macocha returns the following sets: the \( S_{MC} \) set (respectively, \( S_{DC} \)) that contains files that follow exactly the same (Dephase) change pattern and the \( S_{MCH} \) set (respectively, \( S_{DCH} \)) that contains approximate (Dephase) macro co-changing files.

We formulate four research questions:

- **RQ1:** What is the performance of Macocha in the detection of changed files?
- **RQ2:** How does Macocha compare to previous work (Association Rules) in terms of precision and recall?
- **RQ3:** What is the performance, with respect to precision and recall, of Macocha, when detecting occurrences of the Asynchrony pattern?
- **RQ4:** Does Dephase change pattern really exist in practice and if so, how can it be useful?

We perform two types of empirical studies. **Quantitatively**, we compare the findings of Macocha with that of UMLDiff [XS05a] and the co-change analysis of Macocha with the state-of-the-art Association Rules [YMNCC04, ZWDZ04] in term of precision and recall. **Qualitatively**, we use external information provided by bugs reports, mailing lists, and requirement descriptions to validate the Asynchrony and Dephase change patterns not found using previous approaches and to show that these novel change patterns explain real evolution phenomena and thus could help reduce maintenance costs. We apply our approach on seven programs: ArgoUML, FreeBSD, JFreeChart, Openser, SIP, XalanC, and XercesC, developed with three different programming languages, C, C++, and Java.

---

\[http://argouml.tigris.org/issues/show_bug.cgi?id=2926\]
This paper extends our previous work [JGHA11] with the following contributions. First, we describe our approach, Macocha, in more detail and add more discussion about our approach and other related work. Second, we use the \textit{KNN} algorithm to group changes into change periods and therefore to determine automatically the different duration of change periods. Third, we perform an extensive validation of Macocha on three additional case study subjects, JFreeChart, Openser and XercesC, developed with three different languages: C, C++ and Java. In particular, we study the variations in precision and recall of our approach when using different values of its parameters and we perform a static analysis to validate the occurrences of new change patterns. Finally, we provide evidence on the relevance of the Dephase change patterns detected with different shifts in time.

This paper is organised as follows: Section 2 presents Macocha. Section 3 describes our empirical study, while Sections 4 and 5 report and discuss its results as well as threats to validity. Section 6 discusses related work. Section 7 concludes with future work.

2. Our Approach: Macocha

We propose Macocha to mine version-control systems (Concurrent Versions System named CVS$^7$ and Apache Subversion System named SVN$^8$), to identify the change periods in a program, to group source files according to their stability through the change periods, and to identify, among changed files, those that follow the Asynchrony or Dephase change patterns, \textit{i.e.}, are macro co-changing or dephase macro co-changing. We now present the concepts of our approach using examples from ArgoUML.

2.1. Definitions

2.1.1. Change Period

A change period is a period of time during which several commits to different files occurred without “interruption”, \textit{i.e.}, these commits are separated by a few seconds or minutes. Because Macocha use files changelog coming form SVN and CVS system, we use the semantic of commits following the system. For example, for ArgoUML, we analyse the changelog file coming from its SVN system. Thus, we consider that a commit is a set of changes of files that are submitted together (because they have exactly the same date of change). For a program that uses a CVS system, a commit is considered as a singular update to a particular tracked entity having a specific date of change. We conducted a case study to detect the duration of change periods for each subject. We need the concept of change period because the change periods (beginning dates and durations) differ across programs.

Consequently, we want to identify the change periods in a program by grouping all the changes committed closely together in time, independent of the developers who committed them and of their log messages. To identify changes occurring close to one another, \textit{i.e.}, belonging to a same change period, we use the \textit{KNN} algorithm. The \textit{KNN} is a non-parametric learning algorithm [KZP07] that does not make any assumptions on the underlying data distribution.

\footnotesize
$^7$\url{http://cvs.nongnu.org/}
$^8$\url{http://subversion.apache.org/}

Copyright © 2000 John Wiley & Sons, Ltd.
Prepared using \texttt{smrauth.cls}

\textit{J. Softw. Maint. Evol.: Res. Pract.} 2000; 00:1–1
Hatton [Hat07] presented an empirical study to estimate the time for the handling of a particular maintenance request (also known as change request) and showed that the largest duration of a change period to implement a maintenance request is not more than 40 hours. Therefore, we set the initial duration of the change period to 40 hours. However, as we notice earlier: these durations could change across subjects. Thus, we also report the variation of precision and recall using other values for the initial duration of the change period in Section 4. This variation shows us, for the seven subjects in this study, that the 40 hour period yields the best results.

Our KNN-based algorithm to identify the change periods in a program history is illustrated in Figure 3 and consists of four separate phases:

- First, we divide the whole maintenance period, from first to last for the considered changes, of a given program into equal sub-periods. Therefore, following Hatton [Hat07], we divide the history of the program into change periods of equal duration of 40 hours.
- Second, to each change period, we assign the first change committed at a date nearest to, but later than, the beginning date of the change period.
- Third, we use the KNN algorithm with $k = 2$ (we start the execution with $k = 2$ before testing other values of $k$) to assign the rest of the changes into their appropriate change periods: each change is assigned to the change period including its $k$ nearest neighbors in terms of date of commit, whatever their developers and their log messages regardless of commit author and commit log message, and even if its change date is earlier than the beginning date of the change period.
- Fourth, when the KNN algorithm has assigned all changes into change periods, we recompute the beginning and end dates of each change period based on the dates of their earliest and latest changes. If there exists one or more change periods of duration greater than 40 hours, then we reapply the KNN algorithm with an increased value of $k$, else we stop.

Finally, we notice that the KNN algorithm, as any other method, have some advantages and disadvantages [KZP07]. The main advantages of KNN are:

- Very simple implementation.
- Robust with regard to the search space.
- Few parameters to tune.

The main disadvantages of the algorithm are:

- Sensitiveness to noisy or irrelevant attributes, which can be responsible for the large drop in precision at constant recall shown in Section 4.
- Sensitiveness to very unbalanced data sets, where most entities belong to one or a few classes, and infrequent classes are therefore often dominated in most neighborhoods. This can be manifested as some problems to deal with widely varying change periods (from a microsecond to tens of hours).

In Section 4, we show that using this algorithm to group changes into change periods allows us to improve precision and recall over the state-of-the-art Association Rules approach [ZWDZ04] i.e. Macocha has +7% precision, and +4% recall over Association Rules.
In ArgoUML: We find 290 change periods in two years of development. In Figure 4, we present the durations of all the different change periods detected in ArgoUML using the KNN algorithm. The mean duration of these change periods is 27 hours 8 minutes, the standard deviation is 12 hours 6 minutes. In addition, in Figure 4, we use box plots to display differences between the durations of change periods in ArgoUML without making any assumptions concerning the underlying statistical distribution. The spacings between the different parts of the box help indicate the degree of dispersion (spread) and skewness in the data, and identify outliers. In fact, Figure 4 graphically depicts groups of numerical results of ArgoUML through their five-number summaries: the smallest observation (one millisecond),
Figure 4. The distribution of change period durations in ArgoUML detected by the KNN algorithm.

lower quartile (20 hours and 13 minutes), median (29 hours and 16 minutes), upper quartile (35 hours and 17 minutes), and largest observation (39 hours and 58 minutes).

2.1.2. Profile

We define a profile as a bit vector that describes whether a file is changed, or not, during each of the change periods of a program. The length \( n \) of this bit vector is the number of change periods discovered by the KNN algorithm described above.

For each file in a program, its profile is defined as a vector \( x = x_1 \ldots x_n \), where \( n \) represents the number of change periods, i.e. the length of the profile. The value of \( x_i \) indicates whether the file \( F \) is changed or not at the \( i^{th} \) change period:

\[
x_i = \begin{cases} 
1 & \text{if file is changed in the change period } i \\
0 & \text{otherwise.}
\end{cases}
\]

2.1.3. File Stability

Macocha groups files according to their stability: idle and changed, as shown in Figure 5(a). Each group is a set of profiles with similar stability. Idle files do not change after their introduction into the program, while changed files are files that changed after their introduction into the program. Macocha uses this group to identify which changed files follow the Asynchrony or Dephase change patterns.

In ArgoUML: Macocha identifies 1,143 changed files from 1,621 files analyzed in two years of evolution of this program. Indeed, in this program, we detect 478 idle files that are not changed over
DETECTING ASYNCHRONY AND DEPHASE CHANGE PATTERNS

(a) Profiles for change periods of length $n = 10$ showing the changes committed in two different files.

(b) Files F1 and F2 follow the Asynchrony change pattern.

(c) Three different profiles illustrating the Dephase change pattern.

Figure 5. File Stability and Change Patterns.

several versions. As in previous work [LD02], this observation can be explained by the fact that such files can represent a dead code or a good design.

2.1.4. Change Patterns

Similar profiles grouped together represent occurrences of the Asynchrony and Dephase change patterns. Idle files do not change in any change period after their introduction into the program. Therefore, we do not consider this group of files because they are irrelevant to the co-change due to their non-evolution.

Macocha returns the following sets of occurrences of change patterns:

- $S_{MC}$, the set of macro co-changing files with identical profiles in a program;
- $S_{DC}$, the set of dephase macro co-changing files identified when shifting profiles by $s$ change periods with $s \in [0, 5]$;
- $S_{MCH}$, the set of approximate macro co-changing files with similar profiles in a program by using the Hamming distance with $D_H \in [0, 3]$;
- $S_{DCH}$, the set of approximate dephase macro co-changing files identified when shifting profiles by $s$ change periods and by using the Hamming distance with $D_H \in [0, 3]$.
A set $S_{MC}$ contains two or more changed files that exactly change together with long time intervals between their changes and/or performed by different developers and with different log messages, i.e., that have identical profiles during the life of a program, as illustrated in Figure 5(b). Given a file $F_1$, a $S_{DC}$ is the set composed of $F_1$ and one or more files, $F_2...FM$, such that $F_2...FM$ always macro co-change with the same shift in time $s \in [0, n - 1]$ with respect to $F_1$ during the evolution of a program. Thus, a Synchrony change pattern is a Dephase change pattern where $s = 0$.

Figure 5(c) illustrates that $F_1$ and $F_2$ are in dephase macro co-change with $s = 1$; $F_2$ and $F_3$ are in a dephase macro co-change with $s = 2$; and, $F_1$ and $F_3$ are in a dephase macro co-change with $s = 3$. In ArgoUML: Macocha reports that `ProgressEvent.java` and `TestActionAddEnumerationLiteral.java` followed the Dephase change pattern with $s = 2$; and, `ProgressEvent.java` and `IConfigurationFactory.java` followed the Dephase change pattern with $s = 3$. Previous approaches could not detect that these files follow such change patterns.

Macocha considers both identical and similar profiles (with or without shifts in time) to account for cases where the files did not change exactly in the same change periods. We use the Hamming distance [Ham50] $D_H$ to measure the amount of difference between two profiles, *i.e.*, the number of positions at which the corresponding bits are different. For a fixed length $n$, the Hamming distance is a metric on the vector space of the bit vectors of that length, as it fulfills the conditions of non-negativity and symmetry. The Hamming distance between two profiles $a$ and $b$ is equal to the number of ones in the vector $a \oplus b$.

To determine the value of the appropriate Hamming distance for the detection of change pattern occurrences, we conduct a comparative studies based on the evolution of the precision and recall when we use different values of $D_H$, as shown in Figure 6 and Figure 7. In information retrieval contexts, precision and recall are defined in terms of a set of retrieved documents (e.g. the list of co-changed files produced by a query) and a set of relevant documents. Precision is the fraction of co-changed files that are relevant. Recall is the fraction of the co-changed files that are relevant to the query that are successfully retrieved. We also use the F-measure to test accuracy. This measure considers both the precision and the recall of the test to compute the score. The traditional F-measure or balanced F-score (F1 score) is the harmonic mean of precision and recall:

$$F\text{- measure} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

The F-measure score can be interpreted as a weighted average of the precision and recall, where an F-measure score reaches its best value at 1 and worst score at 0. Indeed, we use the F-measure score when we compare the results of Macocha approach and the co-change detection approach based on Association Rules.

We provide more details about the evaluation of our method in Section 4. After analysing several values of $D_H$ between two profiles in different programs, we found that $D_H = 2$ is the best trade-off between precision and recall (F-measure=0.48). Based on this finding, we will consider that two profiles are similar if the Hamming distance between them is equal to 1 or 2. Indeed, Figure 7 shows a set of box plots indicating variability outside the upper and lower quartiles in order to help us to indicate the degree of dispersion (spread) and skewness in the precision and recall, and identify outliers. Figure 7 confirms that $D_H = 2$ is a best trade-off between precision and recall to minimize outliers in the precision and recall of the different subjects. Table I reports the number of occurrences of approximate Asynchrony change pattern detected by Macocha with some values of Hamming distance $D_H$. For
The Hamming distance $D_{H}$

<table>
<thead>
<tr>
<th></th>
<th>$D_{H} = 1$</th>
<th>$D_{H} = 2$</th>
<th>$D_{H} = 3$</th>
<th>$D_{H} = 4$</th>
<th>$D_{H} = 5$</th>
<th>$D_{H} = 6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArgoUML</td>
<td>44</td>
<td>353</td>
<td>439</td>
<td>499</td>
<td>528</td>
<td>621</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>19</td>
<td>98</td>
<td>119</td>
<td>140</td>
<td>239</td>
<td>326</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>116</td>
<td>425</td>
<td>502</td>
<td>533</td>
<td>647</td>
<td>801</td>
</tr>
<tr>
<td>OpenSuse</td>
<td>7</td>
<td>41</td>
<td>50</td>
<td>82</td>
<td>128</td>
<td>166</td>
</tr>
<tr>
<td>SIP</td>
<td>102</td>
<td>570</td>
<td>598</td>
<td>652</td>
<td>703</td>
<td>788</td>
</tr>
<tr>
<td>XalanC</td>
<td>8</td>
<td>69</td>
<td>79</td>
<td>133</td>
<td>142</td>
<td>186</td>
</tr>
<tr>
<td>XercesC</td>
<td>11</td>
<td>125</td>
<td>158</td>
<td>196</td>
<td>206</td>
<td>290</td>
</tr>
</tbody>
</table>

Table I. Number of approximate Asynchrony change pattern occurrences detected using different values of the Hamming distance

Figure 6. The mean of precision and recall achieved by Macocha with different values of $D_{H}$ for the seven programs.

example, we find that with $D_{H} = 4$, Macocha reports 499 occurrences of approximate Asynchrony change pattern in ArgoUML. The precision and recall of Macocha with this value of $D_{H}$ were less than 40%. Therefore, in this paper, we consider that two profiles are similar if their Hamming distance is less than three ($D_{H} < 3$).

Figure 8 illustrates that $F_1$ and $F_2$ are in approximate macro co-change with $D_{H} < 3$; $F_2$ and $F_3$ are in approximate MC with $D_{H} < 3$; and, $F_1$ and $F_3$ are in a approximate MC with $D_{H} < 5$.

In ArgoUML: Macocha reports that $ProgressEvent.java$ and $ProgressListener.java$ follow the Asynchrony change pattern (they have exactly the same profile), and $ProgressEvent-.java$ and $HelpListener.java$ are in approximate MC with $D_{H} < 2$. 

Copyright © 2000 John Wiley & Sons, Ltd.  
Prepared using smrauth.cls
Figure 7. The complete distributions of precision and recall achieved by Macocha with different values of $D_H$ in the form of box plots.

Figure 8. Three different bit vectors showing approximate Asynchrony change pattern

2.2. Data Model, Implementation, and Outputs

A change contains several attributes: the changed file names, the dates of changes, the developers having committed the changes. Using this data, Figure 9 illustrates the concrete process of Macocha. This process takes as input a CVS or SVN change log. First, Macocha calculates the duration of different change periods using the $KNN$ algorithm (1). Second, it groups changes in change periods (2). Third, it creates a profile that describes the evolution of each file in each change period (3). Fourth, it uses these profiles to compute the stability of the files (4) and, then, to identify changed files. Finally, Macocha detects and outputs macro co-changing files, i.e. changed files that follow the Asynchrony
change pattern, and dephase macro co-changing files, i.e. changed files that follow the Dephase change pattern (5).

Macocha also weighs changes according to their distance in time because files co-changing frequently in the past, but not in recent times, may be less interesting than files having recently changed together. To do so, Macocha converts the bit vectors of files following an occurrence of a change pattern to a decimal number. Then, Macocha compares these decimals numbers to report the set of occurrences of change patterns including the most recently changed files (for files changed more recently, the conversion will naturally lead to greater decimal numbers). Ranking occurrences of the Asynchrony and Dephase change patterns allows us to detect the most recent occurrence of change patterns maintained in programs without having any impact on the precision and recall of results. Indeed, for practitioners, the consideration regarding the recency order is weighting changes according to their distance in time: files co-changing frequently in the past but not in recent times are not as interesting as files recently changing together, thus the time should be weighted. Previous work [DLL09] already pointed this problem out and proposed techniques to deal with it. Thus, in this paper, the analysis of change pattern recency does not have any impact on the results in terms of precision, recall and F-measure. However, the concrete examples reported in this empirical study include the most recently changed files.

3. Empirical Study

We use the Goal-Question-Metric (GQM) Approach [BW84] to define our empirical study. Indeed, this approach is an approach to software metrics that defines a measurement model on three levels: conceptual level (goal), operational level (question) and quantitative level (metric). The approach is
based upon the assumption that for an organization to measure in a purposeful way one must first specify the goals for itself and its projects, then one must trace those goals to the data that are intended to define those goals operationally, and, finally, one must provide a framework for interpreting the data with respect to the stated goals.

Following this approach, the goal of our study is to show that Macocha can identify occurrences of the Asynchrony and Dephase change patterns and that these occurrences describe file evolution and co-evolution phenomena. Our purpose is to bring generalisable, quantitative evidence on the existence of the Asynchrony and Dephase change patterns. The perspective is that of both researchers and practitioners who should be aware of the hidden dependencies among files to make informed changes. The context of our study is the maintenance of programs.

3.1. Research Questions

We formulate four research questions:

- **RQ1:** What is the performance of Macocha in the detection of changed files?
  
  We compare the findings of Macocha with that of UMLDiff [XS05a] and we find that Macocha detects the same changed files as UMLDiff, based on a CVS/SVN change log whatever its programming language.

- **RQ2:** How does Macocha compare to previous work (Association Rules) in terms of precision and recall?
  
  We compare the findings of Macocha with the state-of-the-art Association Rules [YMNCC04, ZWDZ04] approach and we find that Macocha improves the identified co-changes in terms of precision and recall.

- **RQ3:** What is the performance, with respect to precision and recall, of Macocha, when detecting occurrences of the Asynchrony pattern?
  
  We use external information provided by bugs reports, mailing lists, and requirement descriptions to validate the Asynchrony change patterns not found using previous approaches.

- **RQ4:** Does Dephase change pattern really exist in practice and if so, how can it be useful?
  
  We provide evidence on the relevance of the Dephase change patterns detected with different shifts in time.

3.2. Case study subjects

We choose seven programs developed with three different programming languages: ArgoUML, JFreeChart and Sip developed with java; FreeBSD and Openser developed with C; XalanC and XercesC developed with C++. Table II summarises program statistics. We use these programs because they are open source, have been used in previous work [CCCDP10a] [ZBLL06], are of different
DETECTING ASYNCHRONY AND DEPHASE CHANGE PATTERNS

Table II. Descriptive statistics of the object programs

<table>
<thead>
<tr>
<th>Languages</th>
<th>ArgoUML</th>
<th>FreeBSD</th>
<th>JFreeChart</th>
<th>Openser</th>
<th>Sip</th>
<th>XalanC</th>
<th>XercesC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numbers of Versions</td>
<td>9</td>
<td>C</td>
<td>Java</td>
<td>C</td>
<td>Java</td>
<td>C++</td>
<td>C++</td>
</tr>
<tr>
<td>Numbers of Files</td>
<td>1,621</td>
<td>500</td>
<td>1,106</td>
<td>5</td>
<td>16</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>Numbers of Commits</td>
<td>6,943</td>
<td>50,145</td>
<td>1,752</td>
<td>5,960</td>
<td>6,100</td>
<td>3,621</td>
<td>3,971</td>
</tr>
<tr>
<td>Numbers of Developers</td>
<td>11</td>
<td>114</td>
<td>4</td>
<td>35</td>
<td>16</td>
<td>11</td>
<td>26</td>
</tr>
</tbody>
</table>

domains and in different programming languages, span several years and versions, and underwent between thousands and tens of thousands of changes.

ArgoUML\(^9\) is an UML diagramming program written in Java and released under the open-source BSD License. We analyse the evolution of this program for a period of two years, from 2007-02-19 to 2009-02-19. In this period, ArgoUML has gone through nine major versions. 11 developers participated in the maintenance of this program by committing 6,943 commits.

FreeBSD\(^10\) is a free Unix operating system written in C and released under the open-source BSD License. We analyse the evolution of this program for a period of two years, from 2007-11-08 to 2009-11-08. In this period, FreeBSD has gone through 11 major versions. We notice that FreeBSD is the program that with the largest number of developers (114) and the largest number of commits (50,145) in this study.

JFreeChart\(^11\) is a Java open-source framework to create complex charts in a simple way. We analyse the evolution of this program from 2008-02-13 to 2010-02-09. In this period, JFreeChart has gone through five versions, and 4 developers maintained its 1,106 files by committing 1,752 commits.

Openser\(^12\) is an open source implementation of a SIP server, licensed under the GNU General Public License. This program can be used as a SIP registrar server, SIP router, SIP redirect server, etc. In addition, it can be used in small programs, for example in embedded programs like DSL routers, but also for large installations at Internet service providers with several million customers. The Openser project was created on 14 June 2005. We detect Asynchrony and Dephase change patterns in this program from this date to March 2007. In this period, Openser has gone through five versions and 35 developers maintained its 383 files by committing 5,960 commits.

SIP Communicator\(^13\) is an audio/video Internet phone and instant messenger that supports some of the most popular VoIP and instant messaging protocols, such as SIP, Jabber, AIM/ICQ, MSN. SIP is open source and freely available under the GNU Lesser General Public License. It is written in Java. We analyse the evolution of this program among 16 versions, from 2006-12-11 to 2008-12-08. In Table II, we notice that this program was maintained by 16 developers.

\(^{9}\)http://argouml.tigris.org/
\(^{10}\)http://www.freebsd.org/
\(^{11}\)http://www.jfree.org/
\(^{12}\)http://www.opensips.org/
\(^{13}\)http://www.sip-communicator.org/
XalanC is an open-source software library from the Apache Software Foundation written in C++. We analyse the evolution of this library for a period of two years, from 1999-02-21 to 2001-12-20. In this period, XalanC has gone through over 13 major versions and 11 developers maintained its 390 files by committing 3,621 commits.

XercesC is a collection of software libraries written in C++ for parsing, validating, serialising, and manipulating XML. We analyse the evolution of this program from its publishing in 99-11-09 to 2001-11-09. In this period, XercesC has gone through 14 versions and 26 developers maintained its 396 files by committing 3,971 commits.

3.3. Analysis

To answer our research questions, we apply Macocha to the different subject programs and we collect all the occurrences of the Asynchrony and Dephase change patterns. We then perform two types of empirical studies. Quantitatively, we first compare in RQ1 the results of Macocha with those of UMLDiff for file stability and we show that Macocha can identify the same idle and changed files as UMLDiff using only data from change logs. It does not produce as detailed information as UMLDiff but this information is sufficient for co-change analysis. Second, we compare in RQ2 the results of Macocha for detecting co-changing files with those of a state-of-the-art approach [ZWDZ04], that uses Association Rules. We also show that the set of macro co-changing files produced by Macocha includes the same co-changing files as reported using Association Rules plus new co-changing files.

Qualitatively, we confirm in RQ3, that each MC found by Macocha but not by the Association Rules-based approach [ZWDZ04] is indeed a dependency link, using external information from static analysis, bug reports, requirement descriptions, and mailing lists.

We also validate in RQ3 each occurrence MC patterns found by Macocha in the analysed programs by studying their static relationships (such as use relations, inheritance relations, and so on). For programs written in Java (ArgoUML, JFreeChart, and SIP), we use an existing tool, PADL [GA08], to automatically reverse-engineer class diagrams from the source code of object-oriented programs. A model of a program is a graph whose nodes are classes and edges are relationships among classes, such as: associations, use relations, inheritance relations, creations, aggregations, and container-aggregations (special case of aggregations [GAA04]). As of today, PADL can only handle all static relations for programs written in Java. For programs considered in this study and written with other languages (FreeBSD, Openser, XalanC, and XercesC), we investigate static relationships among files following the Asynchrony and Dephase change patterns by manual source-code verification.

Indeed, for each (approximate) occurrence of the change pattern, we confirmed in RQ3 and RQ4 the dependencies among their files by detecting their static relationships. If we could not detect such static relationships, we checked for other external information from bug reports and mailing lists. For static analysis, we use automatic tool detection of static relationships among files (Padl). For bug reports and mailing lists, we verify if these files are involving on bugs and mailing list between developers and we discuss the consistency of the external information before confirming the dependencies between these files.

---

14http://xml.apache.org/xalan-c/
15http://xerces.apache.org/xerces-c/
4. Study Results

We now present the results of our empirical study. Table III summarises the cardinalities of the sets obtained by applying Macocha.

4.1. RQ1: What is the performance of Macocha in the detection of changed files?

Macocha groups different commits in programs into change periods detected by the $KNN$ algorithm. First, we observe that for the same duration of maintenance (two years for each program), the number of change periods detected by the $KNN$ algorithm varies between 85 change periods, detected in FreeBSD, and 290 change periods in ArgoUML. Then, we use box plots (illustrated in Figure 10) to display differences between the durations of change periods in the seven analysed programs. We observe that, in each program, change periods detected by the $KNN$ algorithm do not have the same duration. For example, the duration of the change periods detected in JFreeChart varies between 1 millisecond (it is a single commit not clustered with any other commits) and 39 hours 57 minutes. In contrast, Figure 10 shows the distribution of different change periods detected in seven programs. Approximately 75% of the duration of change periods are shorter than 30 hours, so maintenance activities are dominated by small changes measured either by number or total time. We perform an
Figure 11. The variation of precision and recall using different values for the initial duration of change period used by the KNN algorithm.

<table>
<thead>
<tr>
<th></th>
<th>ArgoUML</th>
<th>FreeBSD</th>
<th>JFreeChart</th>
<th>Openser</th>
<th>SIP</th>
<th>XalanC</th>
<th>XercesC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle files</td>
<td>478</td>
<td>302</td>
<td>398</td>
<td>71</td>
<td>314</td>
<td>66</td>
<td>40</td>
</tr>
<tr>
<td>Changed files</td>
<td>1,143</td>
<td>198</td>
<td>708</td>
<td>312</td>
<td>1,379</td>
<td>324</td>
<td>356</td>
</tr>
<tr>
<td># of $S_{MC}$</td>
<td>192</td>
<td>45</td>
<td>281</td>
<td>21</td>
<td>350</td>
<td>41</td>
<td>68</td>
</tr>
<tr>
<td>Max # files</td>
<td>14</td>
<td>12</td>
<td>51</td>
<td>8</td>
<td>36</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>Min # files</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td># of $S_{MC/H}$</td>
<td>353</td>
<td>98</td>
<td>425</td>
<td>41</td>
<td>570</td>
<td>69</td>
<td>125</td>
</tr>
<tr>
<td>Max # files</td>
<td>27</td>
<td>20</td>
<td>61</td>
<td>13</td>
<td>51</td>
<td>11</td>
<td>24</td>
</tr>
<tr>
<td>Min # files</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table III. Cardinalities of the sets of Idle files, Changed files and (approximate) Asynchrony change pattern occurrences obtained in the empirical study

analysis with varying values of the initial duration of change period using by the KNN algorithm to verify if we can prove empirically that, for $t=40$, we get the better precision/recall, and thus, confirm Hatton [Hat07] observation. The variation of precision and recall using other values of the initial duration of the change period is illustrated in Figure 11. A higher value of the initial duration would yield a higher recall but a lower precision. While a smaller value of the initial duration would yield a higher precision but a lower recall.

By analysing each file changed or not during different change periods in programs, Macocha creates the set of profiles describing the evolution of different files in the whole life of the program. This analysis involves also eliminating idle files because they do not change in any change period after their introduction into the program. Therefore, they cannot participate in change patterns (see Table III).

We distinguish idle from changed files by grouping together the files identified as short-lived and active by UMLDiff. Then, we compare the sets provided by UMLDiff and by Macocha and find
that they are identical. For example, as shown in Table IV, Macocha finds 1,143 changed files in ArgoUML, identical to UMLDiff $414 + 729 = 1,143$ short-lived and active files. We note that the finer classification in three clusters of ArgoUML does not have any impact on the co-change analysis.

Table IV reports the number of idle, short-lived, and active files found by UMLDiff in the object-oriented subject programs (ArgoUML, JFreeChart, SIP, XalanC and XercesC) and their categorisation by Macocha. The main limitation for using UMLDiff is that it cannot detect idle, short-lived, and active files in programs developed with non-object-oriented programming languages (FreeBSD and Openser developed in C), because it cannot create their UML-like representations. Machoca improves on UMLDiff in this respect and is able to analyse file stability for any program, providing that their CVS/SVN repositories are available.

Finally, Macocha computes file stability in a few minutes (unlike UMLDiff, which takes a few hours [XS05b]), because it must create UML-like representations of the programs before performing evolution analysis. In the following, we present some examples from the object programs.

In JFreeChart: For two years of maintenance, as shown in Table V, Macocha found 131 change periods. In these periods, we detect 398 idle files. For example, the file ColumnArrangement-.java was modified in only one change period. Using UMLDiff, we confirmed that this file belongs to an idle cluster.

We also detect 708 changed files. For example, the file BarRenderer.java was modified 17 times during the evolution of JFreeChart. Thus, this file belong to the changed group. Using UMLDiff, we confirmed that this file belongs to an active cluster.

<table>
<thead>
<tr>
<th>ArgoUML</th>
<th>Idle Clusters (UMLDiff)</th>
<th>478</th>
<th>Changed Groups (Macocha)</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Short-lived Clusters (UMLDiff)</td>
<td>0</td>
<td>414</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Active Clusters (UMLDiff)</td>
<td>0</td>
<td>729</td>
<td></td>
</tr>
<tr>
<td>JFreeChart</td>
<td>Idle Clusters (UMLDiff)</td>
<td>398</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Short-lived Clusters (UMLDiff)</td>
<td>0</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Active Clusters (UMLDiff)</td>
<td>0</td>
<td>665</td>
<td></td>
</tr>
<tr>
<td>SIP</td>
<td>Idle Clusters (UMLDiff)</td>
<td>314</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Short-lived Clusters (UMLDiff)</td>
<td>0</td>
<td>742</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Active Clusters (UMLDiff)</td>
<td>0</td>
<td>637</td>
<td></td>
</tr>
<tr>
<td>XalanC</td>
<td>Idle Clusters (UMLDiff)</td>
<td>66</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Short-lived Clusters (UMLDiff)</td>
<td>0</td>
<td>122</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Active Clusters (UMLDiff)</td>
<td>0</td>
<td>202</td>
<td></td>
</tr>
<tr>
<td>XercesC</td>
<td>Idle Clusters (UMLDiff)</td>
<td>40</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Short-lived Clusters (UMLDiff)</td>
<td>0</td>
<td>170</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Active Clusters (UMLDiff)</td>
<td>0</td>
<td>186</td>
<td></td>
</tr>
</tbody>
</table>

Table IV. Cardinality of Macocha sets (idle groups and changed groups) in accordance with UMLDiff clusters [XS05a]
In FreeBSD: We find 302 idle files. For example, `kvmproc.c` was modified in one change period in two years.

We also detect 198 changed files. The file `ufsvnops.c` was modified in 15 change periods during the evolution of FreeBSD. We cannot use UMLDiff to verify this result, because UMLDiff cannot analyse file stability in programs written in C.

We answer RQ1: What is the performance of Macocha in the detection of changed files? as follows: Macocha detects the same number of changed files as UMLDiff, in the seven analysed system, based on a CVS/SVN change log.

4.2. RQ2: How does Macocha compare to previous work (Association Rules) in terms of precision and recall?

For each program, Macocha detects files that have identical or similar profiles (the MCs sets) and reports them in Table III. We compare the approximate macro co-changing files found by Macocha with the co-changing files found by an approach based on Association Rules [ZWDZ04] (see also [CCCDP10a]), which uses the Apriori algorithm [AS94] to compute Association Rules. The Apriori algorithm takes a minimum support and a minimum confidence and then computes the set of all Association Rules. To obtain a comprehensive set of rules, we consider as valid rules those achieving a minimum confidence of 0.9 as in previous work [ZWDZ04]. In [ZWDZ05], Zimmermann et al. reported that with this value of confidence, their approach has the best precision and recall. We chose a minimum support of two to compare Association Rules and our approach (because in Macocha, changed files have at least two commits).

We denote the set of co-changing files found by an approach based on Association Rules [ZWDZ04] as $S_{AR}$.

We perform an internal evaluation similar to that of Zimmermann et al.’s [ZWDZ04]. Given snapshots $S_i$, $i \in [1, ..., n]$, we build two sets $T_{train} = \{S_1, ..., S_t\}$ and $T_{test} = \{S_{t+1}, ..., S_n\}$, as shown in Table V. We use $T_{train}$ to build Association Rules and macro co-change dependencies and we compare the co-changing files in $T_{train}$ with those in $T_{test}$. Indeed, Macocha checks if files with similar (or same) profiles in $T_{train}$ have similar (or same) profiles in the $T_{test}$. The reported values for precision and recall are different from the previous study [JGHA11] because the subject programs are analysed for different periods of time.

For the seven programs, we observe that Macocha improves precision and recall over Zimmermann’s approach based on Association Rules, as shown in Table VI. For example, for ArgoUML, results indicate that the precision and the recall of Macocha, respectively 49% and 30%, are better than those of Association Rules, respectively 28% and 29%. For the sum of the objects considered in this study, the improvement in precision is larger than that for recall. Indeed, Macocha has +7% precision and +10% recall over Association Rules, i.e., the precision and the recall of Macocha, respectively 49% and 48%, are better than those of Association Rules, respectively 42% and 38%. Thus, the F-measure value of Macocha, 0.48 is better than the F-measure value of Association Rules, 0.39.

We observed that the Apriori algorithm generates high support sets of rules that are later checked for high confidence. Therefore, high confidence rules with low support are not generated [AS94], which could lead to missed co-changing files.
We answer RQ2: How does Macocha compare to previous work (Association Rules) in terms of precision and recall? as follows: Macocha improves the identified co-changes over an approach based on Association Rules in terms of precision and recall, i.e. Macocha has +7% precision, and +10% recall over Association Rules.

4.3. RQ3: What is the performance, with respect to precision and recall, of Macocha, when detecting occurrences of the Asynchrony pattern?

In the context of the evaluation of co-change analysis approaches, in an internal evaluation [SBA06], we compare groups of co-changing files detected by one approach with change-patterns extracted from a testing set of data that are not involved in the grouping process. An external evaluation [SBA06] is the evaluation of the accuracy of one approach by comparing their results with the result of other approaches or the observation of an expert.

The rationale for an internal evaluation is that no expert, no oracle and no pre-existing groups of co-changing files are available. Precision and recall are measured for the testing sets by considering,
for each file, the groups resulting from the training sets as oracles. Such an internal validation has some limits [VKvRvV09] [DLL09]: (1) Files co-changing frequently in the past (training set) but not recently (test set) will be considered wrongly as false negatives; (2) Files co-changing frequently recently (test set) but not in the past (training set) will be considered wrongly as false positives; (3) If the training set contains false positives or negatives, they cannot be detected using the testing set. In fact, that explain the somewhat low values of precision, recall and F-measure reported in Table VI.

To overcome the limits of an internal validation and to validate change patterns not found using Association Rules, we also performed an external evaluation of Macocha by considering the results of the Association Rules as an oracle and by manually validating the sets for differences between Macocha and Association Rules. Because no expert and no pre-existing groups of co-changing files are available as an oracle, and because the high number of co-changed files detected in the seven subject programs, we chose, first of all, to compare Macocha findings and Association Rules findings. Second, we applied a static analysis to validate the MCs not detected by the approach based on Association Rules because co-change analysis is known to be more useful when combined with static analysis [HH04]. For each (approximate) occurrence of the Asynchrony change pattern, we confirmed the dependencies among their files by detecting their static relationships. If we could not detect such static relationships, we checked for other external information from bug reports, mailing lists, and so on to validate the MCs detected by Macocha.

For each set returned by the Association Rules-base approach, if an identical set was returned by Macocha, it was considered a true positive. If the two sets were not identical, we used external information to validate missing files and to decide if they presented a true positive, a false negative, or a false positive. For example, in JFreeChart, all the sets detected using Association Rules were detected by Macocha except nine sets. We detected static relationships among files of seven sets from the nine missed sets and we confirmed the dependencies among the files of the last two sets using bug reports in the Bugzilla of JFreeChart.

For Xerces, Table VI describes the internal validation, it says that Macocha has +14% in precision, and +10% in recall over Association Rules. It means, for the testing set, Macocha has less false negatives and false positives than Association Rules. In Table VII, Macocha has 100% precision and recall versus Association Rules. It means that Macocha and Association Rules detected exactly the same sets in the training set. Table VII also reports, under the Association Rules header, the precision and recall of Macocha with respect to the approach based on Association Rules [ZWDZ04]. The precision and recall presented in Table VII are measured relative to another method. So, precision really means “number of cases that Macocha said were true/false, that also Association Rules said were true/fals”, and recall really means “number of cases that Association Rules found, that also Macocha found”. Table VII shows that Macocha detects the majority of co-changing files detected using Association Rules in the seven object programs. In addition, Macocha detects other occurrences of change patterns not detected using Association Rules.

Table VIII and Table VIII report, under the External Information header, the adjusted precision and recall values of Macocha after manual validation, which show that Macocha detects occurrences of change patterns missed or wrongly reported using Association Rules. For example, in Openser, 89% of co-changing files found by Macocha were detected by Association Rules (the value of precision in Table VII). While 95% of co-changing files detected by Association Rules were detected by Macocha (the value of recall in Table VII). This comparison gives us 14 cases of false positives and 5 cases of false negatives. We confirmed all of these cases, as shown in Table VIII and Table IX, by a static
analysis of source code of these files performed manually. We do not obtain 100% recall because that our results are affected by the presence of false negatives when using the results of an approach based on Association Rules [ZWDZ04] as oracle. This fact is due to the differences between an approach that makes predictions at the level of commits and one at the level of change periods. Indeed, a smaller value of change period duration would yield a higher recall but without detecting novel change patterns such as Asynchrony change pattern because we do not integrate the analysis of files that are maintained by different developers and–or with some delay in time, which could lead to missed co-changing files and change propagation scenarios.

In the following, we describe some (approximate) occurrences of the Asynchrony change pattern that are missed by the previous approach and justify why they are missed and why it is important to detect them. We chose these examples from the most-recently changed files following the (approximate) Asynchrony change pattern.

**In ArgoUML:** SelectionActionState.java and SelectionState.java followed the same occurrence of an Asynchrony change pattern. On the one hand, by using PADL [GA08] to automatically reverse-engineer class diagrams from the source code of ArgoUML, we detected a static dependency among these two files. On the other hand, in the Bugzilla of ArgoUML, the bug ID 2552 states that “we should use 3 state [...] this model could simply be the Action” in relation with these two files. By applying the co-change analysis for “Error Prevention” described in [ZWDZ04], we could not find co-change dependencies among them. Thus, we could not explain and–or predict bugs in relation to these two files.

16 [http://argouml.tigris.org/issues/show_bug.cgi?id=2552](http://argouml.tigris.org/issues/show_bug.cgi?id=2552)
Table VIII. Adjusted precision of Macocha when using the results of an approach based on Association Rules [ZWDZ04] as oracle and after manual validation using external information and static analysis (V.S.A: Validation by static analysis)

<table>
<thead>
<tr>
<th></th>
<th>False positives</th>
<th>V.S.A</th>
<th>Bugs</th>
<th>Mails</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArgoUML</td>
<td>11</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>100%</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>16</td>
<td>10</td>
<td>0</td>
<td>4</td>
<td>88%</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>13</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>93%</td>
</tr>
<tr>
<td>Openser</td>
<td>14</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>Sip</td>
<td>12</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td>100%</td>
</tr>
<tr>
<td>XalanC</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>XercesC</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table IX. Adjusted Recall of Macocha when using the results of an approach based on Association Rules [ZWDZ04] as oracle and after manual validation using external information and static analysis (V.S.A: Validation by static analysis)

<table>
<thead>
<tr>
<th></th>
<th>False negatives</th>
<th>V.S.A</th>
<th>Bugs</th>
<th>Mails</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArgoUML</td>
<td>56</td>
<td>49</td>
<td>2</td>
<td>4</td>
<td>99%</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>10</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>100%</td>
</tr>
<tr>
<td>Openser</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>Sip</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>XalanC</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>100%</td>
</tr>
<tr>
<td>XercesC</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
</tbody>
</table>

**JFreeChart**: AbstractXYDataset.java and RenderAttributes.java were in MC. This is confirmed in the Bugzilla of JFreeChart by the bug ID 1654215\(^{17}\) relating these two files. In fact, this bug reports that “Adding renderer with no dataset causes exception” and confirms static dependencies detected after analysing JFreeChart source code by PADL. These two files were changed by the same developer in a time-window of more than a few minutes. Therefore, by applying the

\(^{17}\)http://sourceforge.net/tracker/index.php?func=detail&aid=1654215&group_id=154949&atid=115494
Association Rules-based approach described in [ZWDZ04], we could not not find that these files were co-changing. Consequently, we could not give to the developer the knowledge about dependencies among these two files to maintain them properly, as described in [YMNCC04, ZWDZ04].

We answer RQ3: What is the performance, with respect to precision and recall, of Macocha, when detecting occurrences of the Asynchrony pattern? as follows: Macocha detects change patterns missed or wrongly reported using Association Rules. Concretely, 97% of co-changing files found by Macocha were also detected by Association Rules, while 90% of co-changing files detected by Association Rules were also detected by Macocha. This comparison provided us with 66 cases of false positives and 73 cases of false negatives.

4.4. RQ4: Does Dephase change pattern really exist in practice and if so, how can it be useful?

No previous approach could detect files maintained with similar trends and some given shifts in time. Indeed, Dephase change pattern is the main contribution of the paper in term of novelty. We confirmed the existence of occurrences of the (approximate) Dephase change patterns by detecting static relationships among their files. We also confirmed these occurrences using external information. Table X illustrates the number of occurrences of the (approximate) Dephase change pattern detected and confirmed using external information and static analysis, in each program. We recall that a Synchrony change pattern is a Dephase change pattern with \( s = 0 \).

Macocha can detect occurrences of the (approximate) Dephase change pattern with several values of shift \( s \). After analysing different sets of dephase macro co-changing files detected in the seven object programs, we observed that the number of occurrences of the (approximate) Dephase change pattern detected by Macocha and confirmed by external information decreased from \( s = 3 \) and is close to 0 from \( s = 5 \), as shown in Table X. In our case study, we detected dephase macro co-changes for \( s \in [1, 5] \) to obtain an accurate sets of results. In Table X we note that, for example, out of the 27 occurrences of Dephase change patterns detected in ArgoUML with a shift \( s=2 \), 24 occurrences were confirmed by static analysis. In our case study, the precision was 88%. As other examples derived from Table X, we note the small number of (approximate) Dephase change pattern occurrences detected in programs developed in C or C++ (less than six occurrences for each value of the shift \( s \)).

We now report some typical occurrences of the (approximate) Dephase change pattern from different programs with different values of shift \( s \). We chose these examples from the most-recently changed files following the (approximate) Dephase change patterns.

**In FreeBSD:** We find that `ip-fw2.c` and `sysv-msg.c` follow the same occurrence of a Dephase change pattern with a shift \( s = 2 \). After manually investigation of the source code of these two files we do not find any static relationships among them. However, in the mailing list of FreeBSD, the Message-ID: `<20120107201823.H3704@sola.nimnet.asn.au>` states that the two files were used to implement the same requirement of “ruleset sequence” in a lengthy message from a developer about “IPFW transparent VS dummynet rules”. Our case study confirms dependencies among these two files by external information.

**In SIP:** We find that `Html2Text.java` and `FileTransfReceiveListener.java` were changed systematically with five shift periods in two years. Therefore, they followed the Dephase
change pattern with a shift $s = 5$. These two files implement the same feature\(^{18}\); “Instant Messaging”. By performing a static analysis, we detected a static dependency among these two files. Therefore, we confirmed the occurrence of the Dephase change pattern formed by these two files.

\(^{18}\)http://www.jitsi.org/index.php/Main/Features

<table>
<thead>
<tr>
<th>Shift</th>
<th>$s = 1$</th>
<th>$s = 2$</th>
<th>$s = 3$</th>
<th>$s = 4$</th>
<th>$s = 5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArgoUML</td>
<td>Approximate DC</td>
<td>46</td>
<td>27</td>
<td>39</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>Approximate DC confirmed</td>
<td>32</td>
<td>24</td>
<td>30</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>DC</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>DC confirmed</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>Approximate DC</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Approximate DC confirmed</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>DC</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>DC confirmed</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>Approximate DC</td>
<td>46</td>
<td>14</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Approximate DC confirmed</td>
<td>43</td>
<td>14</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>DC</td>
<td>6</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>DC confirmed</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Openser</td>
<td>Approximate DC</td>
<td>1</td>
<td>8</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Approximate DC confirmed</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>DC</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>DC confirmed</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Sip</td>
<td>Approximate DC</td>
<td>43</td>
<td>68</td>
<td>89</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td>Approximate DC confirmed</td>
<td>43</td>
<td>66</td>
<td>80</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>DC</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>DC confirmed</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>XalanC</td>
<td>Approximate DC</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Approximate DC confirmed</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>DC</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>DC confirmed</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>XercesC</td>
<td>Approximate DC</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Approximate DC confirmed</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>DC</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>DC confirmed</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table X. Evolution of the number of occurrences of (Approximate) Dephase change patterns, DC, detected and manually confirmed by static analysis for different values of shift $s$. 

\cite{http://www.jitsi.org/index.php/Main/Features}
In XercesC: We found that XercesXPath.cpp and XMLDateTime.cpp follow the same occurrence of an approximate Dephase change pattern with shift $s = 1$. This change dependency is confirmed by multiple static relationships detected when we examine the source code of these two files. In addition, in the mailing list of XercesC, a message\(^{1}\) on April 1, 2009 about “a legitimate bug with the time of day” states that these two files are related.

Indeed, our approach guides programmers based on the program history. Suppose the developer changed a file F1 in the program. Macocha then suggests to change the file F2 because in the past, both items always have been changed together with same shift in time. All Macocha needs is a CVS/SVN repository. The benefit of Macocha is that it points out item coupling that is undetectable by previous co-change analysis such as between files maintained by different developers or—and with some shift in time. In the following, we show how Dephase macro co-changing files detected using Macocha support the following three scenarios:

4.4.1. Scenario 1: Management of Development Teams

We think that if two files follow the same occurrence of a Dephase change pattern, they probably should be maintained by the same team of developers to minimise the risks of introducing bugs in the future. Otherwise, if it is an obligatory to have different team maintained files follow the same occurrence of a Dephase change pattern (for example in the case where a feature is developed by a product team and tests for that feature are developed by the test team), these teams should exchange information about these files after each change. We notice that they might probably send each other private emails notifying the other party of a change. In our context, it is neither possible to conclude that developers are sending each other private emails notifying the other party, nor is the opposite true. The team of developers most likely possesses a wealth of unwritten knowledge about the design and implementation choices that they made for these files, which would help them to prevent introducing bugs [RWBW90].

Consequently, a team leader could redefine the organisation of the maintenance team according to the Dephase macro co-changes links among files, so that her team does not introduce bugs because of the absence of information or lack of communication among developers. For example, in ArgoUML, when we analysed changes made in three Dephase macro co-changing files that have generated bugs (BugID 1957 BugID 2926, and BugID 4604), we found that these changes have been made with one shift in time in their periods of change and by different developers. These co-changes cannot be detected by previous co-change analysis approaches. Thanks to Dephase macro co-change, a team leader could ensure that the team who will maintain these files in each change period has the necessary knowledge to maintain the dependency among these files.

4.4.2. Scenario 2: Bug and Change Propagation

If co-change dependencies are not properly maintained, developers could introduce bugs to a program [DLR09]. Knowing that two files are in Dephase macro co-change implies the existence of (hidden)

\(^{1}\)http://markmail.org/message/a5secbiwkgxtexgb
co-change dependencies between these two files. With our approach, for each program studied, we detected files in Dephase macro co-changes. By using external information, we confirmed our observation and that some of these files indeed participate to bugs. For example, in SIP, we detected seven bugs in relation with Dephase macro co-changing files. By applying the association rule approach described in [ZWDZ04], we could not find that these files are co-changing. A full list of defects belonged to Synchrony and Dephase change patterns detected in the seven analysed programs is available on-line at http://www.ptidej.net/downloads/experiments/jsme12/. Therefore, by knowing files that are in DMMCs, Macocha provides the list of files that they should be carefully considered by developers to ensure the change propagation and the proper maintainability.

4.4.3. Scenario 3: Traceability Analysis

The change history represents one of sources of information available for recovering traceability links that are manually created and maintained by developers [KM07]. The version history may reveal hidden links that relate files and would be sufficient to attract the developers’ attention [KMS07]. For example, in SIP, we detect traceability links between four approximate Dephase macro co-changing files. By applying the association rule approach described in [ZWDZ04], we could not find that these files are co-changing.

Due to the distributed collaborative nature of open-source development, version-control systems are the primary location of files and the primary means of coordination and archival [KMS07]. The requirements of open-source programs are typically implied by communication among project participants and through test cases. However, such traces of requirements are lost in time. In a previous approach [AGA11], we presented an approach, Histrace, that used CVS/SVN change logs to build traceability links between high-level documentation and source code entities, observing that log messages are tied to changed entities and, thus, can be used to infer traceability links. Indeed, Histrace improved with statistical significance the precision of the traceability links [AGA11], while also improving recall but without statistical significance. We thus showed that our trust-based approach indeed improves precision and recall and also that CVS/SVN change logs are useful in the traceability recovery process. Ongoing work includes using the (approximate) Dephase macro co-change to improve traceability links detection between files in the same system.

We answer RQ4: Does Dephase change pattern really exist in practice and if so, how can it be useful? as follows: in all object programs, Macocha detects occurrences of the (approximate) Dephase change pattern, e.g., 183 occurrences were detected in ArgoUML, while these occurrences specify change propagation scenarios as well as they can help to reorganize maintenance tasks by spotting hidden dependencies among files.

5. Discussions

In RQ1, we reported that Macocha can identify changed files before performing the co-change analysis. On the one hand, we showed that Macocha was able to analyse file stability for any program, providing that their CVS/SVN repositories are available. On the other hand, Macocha computed file stability in a few minutes (unlike UMLDiff, which takes few hours [XS05b]). It could be true that a baseline
approach (a file is idle if it never changes 40 hours after its introduction) could perform pretty well and report similar results without needing the additional complexity. However, with a baseline approach, we could not apply the other steps of our approach that allow for the detection of new change patterns: Asynchrony and Dephase change patterns.

Indeed, we performed the first step to eliminate files that do not correspond to a meaningful modification task.

A major application for co-change detection approaches is to guide users through source code. The user changes some entity and these approaches recommend possible future changes in a view. To evaluate the predictive power of a co-change detection approach in this situation, the authors of a previous approach based on Association Rules [ZWDZ04] tested the capability of their approach to identify future changes. For each transaction T, and each entity e belong to entities(T), they queried Q = e, and checked whether the approach would identify E = entities(T) - e. For each transaction, they thus tested entities(T) queries, each with one element. We repeat the same analysis in RQ2. In fact, Zimmermann et al. [ZWDZ04] applied Association Rules to identify co-changing files and showed that increasing the support threshold also increases the precision, but decreases the recall as their approach gets more cautious. However, using the highest possible thresholds does not always yield the best precision and recall values. If they increased the confidence threshold above 0.80, both precision and recall decrease. Furthermore, Zimmermann et al. showed that a high support and confidence threshold is required for high precision. Still, such values result in a very low recall, indicating a trade-off between precision and recall. In our study conducted in this paper, we showed that their approach can identify 38% of all entities changed later in the same transaction. While, Macocha can identify 48% of all entities changed later in the same transaction for the same programs. Approaches based on Association Rules compute only the frequency of co-changed files on individual commits and omit many other cases, e.g., files that co-change with some shifts among change periods.

In RQ3 and RQ4, we showed some gains of our approach compared to previous co-change analysis approaches. For example, approaches based on Association Rules cannot detect all occurrences of co-change and any occurrences of Dephase macro co-changes because, by their very definition, they do not integrate the analysis of files that are maintained by different developers and--or with some shift in time, which could lead to missed co-changing files and change propagation scenarios.

While the main contribution of our work is detecting several occurrences of the (approximate) Asynchrony and Dephase change patterns (two novel change patterns) in different programs belonging to different domains and with different sizes, histories, and programming languages. However, we do not detect macro co-changing files and Dephase macro co-changing files with the same proportions in each program. We observe that the numbers of Macro co-changes and Dephase macro co-changes found in the programs developed in Java (ArgoUML, JFreeChart, and SIP) are greater than the number of Macro co-changes and Dephase macro co-changes found in programs developed in C or C++ (see Table III and Table X. However, we observe that, on the one hand, the majority of FreeBSD files are idle and that, on the other hand, Openser, XalanC, XercesC are the smallest object programs (having less than 400 files). Indeed, we apply our approach to detect (Dephase) macro co-changes on fewer C and C++ files than Java files, which could explain the lower number of Macro co-changes and Dephase macro co-changes. In future work, we will conduct studies on other programs in these languages to assess the numbers of Macro co-changes and Dephase macro co-changes according to the programming languages.
5.1. Threats to Validity

Some threats limit the validity of the results of our empirical study.

**Construct Validity:** Construct validity threats concern the relation between theory and observations. In this study, they could be due to implementation errors. They could also be due to a mistaken relation between changed files. We believe that this threat is mitigated by the facts that many authors discussed this relation, that this relation seems rational, and that the results of our analysis shows that, indeed, Macro co-changes and Dephase macro co-changes exist and are corroborated by external sources of information (bug reports and others). In addition, our results can be affected by the presence of false negatives, i.e., by a low recall exhibited by the co-change detection approach. As previous work detected co-changes committed by the same author in a short time window, relaxing these constraints may also lead to false positives. The results of our empirical study show that Macocha improves precision and recall with respect to the state of the art in seven different programs. However, we cannot claim that our approach will give similar results for any program.

**Internal Validity:** Internal validity is the validity of causal inferences in studies based on experiments. The internal validity of our study is not threatened because we have not manipulated a variable (the independent variable) to see its effect on a second variable (the dependent variable).

**Reliability Validity:** Reliability validity threats concern the possibility of replicating this study. We attempted to provide all the necessary details to re-implement our approach and replicate our empirical study. Moreover, both ArgoUML, FreeBSD, JFreeChart, Openser, Sip, XalanC, and XercesC source code repositories are publicly available. the way our analysis were performed is described in detail in Section 2. The change logs, the list of bugs and the changed files of the seven programs analysed with their profiles to obtain our observations are available on-line at [http://www.ptidej.net/downloads/experiments/jsme12/](http://www.ptidej.net/downloads/experiments/jsme12/).

**External Validity:** We performed our study on seven different real programs belonging to different domains and with different sizes, histories, and programming languages. Yet, we cannot assert that our results and observations are generalisable to any other programs, and the fact that all the analysed programs are open-source may reduce this generability. Nevertheless, it would be desirable to analyze further systems, also developed in different programming languages, to draw more general conclusions. Future work includes replicating our study in other contexts and with other programs.

6. Related Work

The concepts of Macro co-changes and Dephase macro co-changes relate our work to that on file stability, co-change and change patterns, and change propagation. We present and discuss related work in comparison to our approach.

6.1. File Stability

Many approaches exist to group files based on their relative stability throughout the software development life cycle. For example, UMLDiff [XS05a] compares and detects the differences between the contents of two object-oriented program versions. A fact extractor parses each version to extract models of their design. Next, a differencing algorithm, UMLDiff, extracts the history of the program
evolution, in terms of the additions, removals, moves, renamings, and signature-changes of design entities, such as packages, classes, interfaces, and their fields and methods. UMLDiff then assigns a stability to each class: short-lived classes (that exist only in a few versions of the program and then disappear), idle classes (that rarely undergo changes after their introduction in the program), and active classes (that keep being modified over their whole lifespan).

Kpodjedo et al. in [KRGA09] and [KRG+10] proposed to identify all files that do not change in the history of a program, using an Error Tolerant Graph Matching algorithm. They studied the evolution of the program class diagram by collecting program source code over several years, reverse-engineering their class diagrams, and recovering traceability links between subsequent class diagrams. Their approach identified evolving classes that maintain a stable structure of relations (association, inheritance, and aggregation) and so on, which likely constitute the stable backbone of a program.

Lanza et al. [LD02] presented an evolution matrix to display the evolution of the files of a program. Each column of the matrix represents a version of the program, while each row represents the different versions of the same file. Within the columns, the files are sorted alphabetically. Then, the authors presented a categorisation of files based on the visualisation of different versions of a class: a pulsar class grows and shrinks repeatedly during its lifetime, a supernova file suddenly explodes in size, a white dwarf is a file that used to be of a certain size, but lost its functionality, a red giant file tends to implement too many functionalities and is quite difficult to refactor, and an idle file does not change over several versions.

Discussion:
The Error Tolerant Graph Matching algorithm and UMLDiff require parsing and comparing AST-like representations of the programs before performing their analysis. We propose to compute stability more simply using the version control systems, which keeps track of all work and all changes in each file in the program.

Our work differs in the level of granularity and on the aspects considered. Indeed, Lanza et al. [LD02] considered only file implementation to identify stability in different versions without considering information coming from version-control systems. Therefore, idle classes for example are those that did not change too much after their introduction in terms of source code and not in terms of commits. Thus, if a file changes frequently but without major modifications in it implementation during the observation period, it will be identified as an “idle file”, which is contradictory to its category name (idle files). This, we propose to identify file stability by mining program history. In the context of change analysis, if a file changes frequently, it will be identified as a “changed file”.

6.2. Co-changing Files

Ying et al. [YMNCC04] and Zimmermann et al. [ZWDZ04] applied Association Rules to identify co-changing files. Their hypothesis is that past co-changed files can be used to recommend source code files potentially relevant to a change request. An association-rule algorithm extracts frequently co-changing files of a transaction into sets that are regarded as change patterns to guide future changes. Such an algorithm uses co-change history in CVS and avoids the source code dependency parsing process.

Ceccarelli et al. [CCCDP10b] and Canfora et al. [CCCDP10a] proposed the use of a vector auto-regression model, a generalisation of univariate auto-regression models, to capture the evolution and the inter-dependencies between multiple time series representing changes to files. They used the bivariate
Granger causality test to identify if the changes to some files are useful for forecasting changes to other files. They concluded that the Granger test is a viable approach to change impact analysis and that it complements existing approaches like Association Rules to capture co-changes.

Antoniol et al. [ARV05] presented an approach to detect similarities in the evolution of files starting from past maintenance. They applied the LPC/Cepstrum technique, which models a time evolving signal as an ordered set of coefficients representing the signal spectral envelope, to identify in version-control systems the files that evolved in the same or similar ways. Their approach used cepstral distance to assess series similarity (if two cepstra series are “close”, the original signals have a similar evolution in time).

Bouktif et al. [SBA06] defined the general concept of change patterns and described one such pattern, Synchrony, that highlights co-changing groups of artefacts. Their approach used a sliding window algorithm as in [ZWDZ04] to build Synchrony change pattern occurrences.

Discussion:
Approaches based on Association Rules [YMNCC04], [ZWDZ04], compute only the frequency of co-changed files in the past and they omit many other cases, e.g., files that co-change with some shifts among change periods. In Section 4, we showed that approaches based on Association Rules cannot detect all occurrences of co-change and any occurrences of Dephase macro co-changes because, by their very definition, they do not integrate the analysis of files that are maintained by different developers and–or with some shift in time, which could lead to missed co-changing files.

Indeed, previous approaches could find files having very similar maintenance evolution history, but they did not present a tool to detect co-changed files maintained by different developers in periods of time more than a few minutes. We were inspired by [SBA06] to name the Asynchrony change pattern presented in this paper. We also introduced a novel change patterns inspired from co-changes, the Dephase change pattern, that describes co-change among files with some shift in time.

6.3. Change Propagation

Change propagation analyses how changes made to one file propagate to others. Law and Rothermel [LR03] presented an approach for change propagation analysis based on whole-path profiling. Path profiling is a technique to capture and represent a program dynamic control flow. Unlike other path-profiling techniques, which record intra-procedural or acyclic paths, whole-path profiling produces a single, compact description of a program’s control flow, including loops iteration and inter-procedural paths. Law et al.’s approach builds a representation of a program’s behavior and estimates change propagation using three dependency-based change-propagation analysis techniques: call graph-based analysis, static program slicing, and dynamic program slicing.

Hassan and Holt [HH04] investigated several heuristics to predict change propagation among source code files. They defined change propagation as the changes that a file must undergo to ensure the consistency of the program when another file changed. They proposed a model of change propagation and several heuristics to generate the set of files that must change in response to a changed file.

Zhou et al. [ZWG+08] presented a change propagation analysis based on Bayesian networks that incorporates static source code dependencies as well as different features extracted from the history of a program, such as change comments and author information. They used the Evolizer system that retrieves all modification reports from a CVS and uses a sliding window algorithm to group them.
Canfora and Cerulo [CC05] proposed an approach to derive the set of files impacted by a proposed change request. A user submits a new change request to a Bugzilla database. The new change request is then assigned to a developer for resolution, who must understand the request and determine the files of the source code that will be impacted by the requested change. Their approach exploits information retrieval algorithms to link the change request descriptions and the set of historical source file revisions impacted by similar past change requests.

D’Ambros et al. [DLL09] presented the Evolution Radar, an approach to integrate and visualise module-level and file-level logical couplings, which is useful to answer questions about the evolution of a program, the impact of changes at different levels of abstraction, and the need for restructuring. German [Ger06] used the information in the CVS to visualize what files are changed at the same time and who are the people who tend to modify certain files. He presented SoftChange, a tool that uses a heuristic based on a sliding window algorithm to rebuild the Modification Record (MRs) based on file revisions. In Softchange, a file revision is included in a given MR if all the file revisions in the MR and the candidate file revision were created by the same author and have the same log. Beyer and Hassan [BH06] introduced the evolution story-board, a new concept for animated visualisations of historical information about the program structure, and the story-board panel, which highlights structural differences between two versions of a program. They also formulated guidelines for the usage of their visualisation by non-experts and to make their evaluations repeatable on other programs.

Discussion: We share with all the above authors the idea that change propagation identification into existing source code is a powerful mechanism to assess code maintainability. Their change-propagation models can be used to predict future change couplings, but they do not allow differentiation between different change patterns. All of these approaches grouped change couplings created by the same author and have the same log message; consequently, they cannot detect typical situation of co-changed file such as file maintained by different developers.

In addition, Xing and Stroulia [XS07], reported that visualisations approaches are limited in their applicability, because they assume a substantial interpretation effort by their users and they do not scale well: they become unreadable for large programs with numerous components.

7. Conclusion and Future Work

The development and maintenance of a program involves handling large numbers of files. These files are logically related to each other and a change to one file may imply a large number of changes to various other files. Many previous works try to reduce program maintenance costs by detecting and using co-changing files. For example, the authors in [SBA06] defined the Synchrony change pattern as common and recurring modifications of programs’ files in time.

In this paper, we introduced the Asynchrony change pattern and the Dephase change pattern, as well as their approximate versions, to explain other scenarios of co-change and change propagation, which could help developers to maintain a program’s files appropriately. We proposed an approach, Macocha, which mines software repositories and uses several algorithms and techniques, such as the $KNN$ algorithm, the Hamming distance, and a bit vector model, to discover occurrences of the (approximate) Asynchrony and Dephase change patterns.

Macocha relates to file stability and co-changes. We therefore performed two types of empirical studies. Quantitatively, we compared Macocha with UMLDiff [XS05a] and an Association Rules-
based approach [ZWDZ04] by applying and comparing the results of the three approaches on seven programs: ArgoUML, FreeBSD, JFreeChart, Openser, SIP, XalanC, and XercesC, and we showed that Macocha has better precision and recall than the state-of-the-art approaches based on Association Rules [YMNC04, ZWDZ04]. Qualitatively, we used external information and static analysis to show that detected Macro co-changes and Dephase macro co-changes explain real, important evolution phenomena. We also showed that occurrences of Dephase change patterns do exist and can help in explaining bugs, managing development teams, and performing traceability analysis.

Therefore, this paper extended our previous work [JGHA11] with the following contributions. First of all, we confirmed the existence of two novel change patterns in seven programs (three more programs: JFreeChart, Openser and XercesC) developed with three different languages: C, C++, and Java. Second, we generalize our previous approach by using the KNN algorithm to group changes into change periods and therefore to determine automatically the duration of the different change periods in each program. Third, we studied the variations in precision and recall of our approach when using different values of its parameters (the Hamming distance, the number \( s \) of shifting profiles, and the start parameter for the detection of change periods).

We are currently (1) relating change patterns with design patterns, (2) identifying other scenarios in which Asynchrony and Dephase change patterns help in reducing maintenance costs, (3) evaluating the consistency and the usefulness of change patterns’ occurrences, including files recently changed over other occurrences, (4) relating Asynchrony and Dephase change patterns to program quality and external software characteristics, such as change proneness. Future work also includes empirical studies of the usefulness for developers of ranking occurrences of the Asynchrony and Dephase change patterns as well as applying Macocha to different C/C++ programs.

Acknowledgements

This work has been partly funded by a FQRNT team grant, the Canada Research Chair in Software Patterns and Patterns of Software. We gratefully thank Massimiliano Di Penta and Daniel M. German for their generous comments.

REFERENCES


