An Exploratory Study of Macro Co-changes

Fehmi Jaafar[1], Yann-Gaël Guéhéneuc[1], Sylvie Hamel[2], and Giuliano Antoniol[1]
[1] PTIDEJ Team, SOCCER Lab, DGIGL, École Polytechnique de Montréal, QC, Canada
[2] LBIT Team, DIRO, Université de Montréal, QC, Canada
E-Mails: {jaafarfe,hamelsyl}@iro.umontreal.ca, yann-gael.gueheneuc@polymtl.ca, antoniol@ieee.org

***Abstract*—The literature describes several approaches to identify the artefacts of programs that change together to reveal the (hidden) dependencies among these artefacts. These approaches analyse historical data, mined from version control systems, and report co-changing artefacts, which hint at the causes, consequences, and actors of the changes. We introduce the novel concepts of *macro co-changes* (MCC), *i.e.*, of artefacts that co-change within a large time interval, and of *dephase macro co-changes* (DMCC), *i.e.*, macro co-changes that always happen with the same shifts in time. We describe typical scenarios of MCC and DMCC and we use the Hamming distance to detect approximate occurrences of MCC and DMCC. We present our approach, Macocha, to identify these concepts in large programs. We apply Macocha and compare it in terms of precision and recall with UMLDiff (file stability) and association rules (co-changing files) on four systems: ArgoUML, FreeBSD, SIP, and XalanC. We also use external information to validate the (approximate) MCC and DMCC found by Macocha. We thus answer two research questions showing the existence and usefulness of theses concepts and explaining scenarios of hidden dependencies among artefacts.**

***Keywords*—Co-changes; stability; bit vectors.**

I. Introduction

Developers must continually change their software programs to meet new requirements and user needs, else their programs become progressively unsatisfactory [1] and eventually become obsolete to the point of disappearing. The literature describes many approaches to extract and analyse the changes undergone by software artefacts and to infer patterns that describe these changes to help program comprehension and evolution. Several of these approaches identify co-changes among artefacts, *e.g.*, [2], [3], which represent the (often implicit) dependencies or logical couplings among artefacts that have been observed to frequently change together [4]. Two artefacts are co-changing if they were changed by the same author and with the same log message in a time-window of less than 200 ms. [3]. Mockus *et al.* [5] defined the proximity in time of checkins by the check-in time of adjacent files that differ by less than three minutes. Other studies (*e.g.*, [6] and [7]) described issues about identifying atomic change sets and reported that, in all cases, they differed by few minutes.

Artefacts can be source code files, classes in object-oriented programs, specifications, and so on. In this paper, as in previous work, *e.g.*, [3], [8], and [9], for the sake of simplicity, we focus on C, C++, and Java source files (.c, .cpp, and .java) as they are among the most common and popular programming languages.

Previous co-changes are intrinsically limited in time. They cannot express patterns of changes between long time intervals. For example, in the Bugzilla of ArgoUML, the bug ID 5378[1] 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 thus confirms that these two files are related. However, no previous approach can detect that these files co-changed because they were maintained by the same developper `bobtarling` but their changes were separated by few hours. Yet, Knowing the dependency among theses files is useful to a new developer that must change `ArgoDiagram.java`: she must assess `ModeCreateAssociationClass.java` for change.

In general, this scenario happens 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 2. This scenario may repeat for years and would be undetected using a sliding window of few minutes. Yet, it contains important information both for the developer and her colleagues: changes to `F1` must likely propagate to `F2`.

As another example in ArgoUML, we found that the developers `mvw` and `tfmorris` contributed with some patches that contains `NotationUtilityJava.java` and `ModelElementNameNotationUml.java`[2] and the bug ID 2926[3] confirms that the two files are related (see Section IV for details). No previous approach can detect that these files co-changed because , during the 11 years of development of ArgoUML, these two files were never changed by the same developer at the same time but were changed by developers `mvw` and `tfmorris` in two consecutive change periods: first `NotationUtilityJava.java` and, subsequently after one period of change, `ModelElementNameNotationUml.java`, pointing out dependency among

[1] http://argouml.tigris.org/issues/show_bug.cgi?id=4604

[2] http://argouml.tigris.org/issues/showattachment.cgi/2118/20101116-patch-notation.txt

[3] http://argouml.tigris.org/issues/show_bug.cgi?id=2926

these two files.

In fact, this scenario can happen when a developer D2 is always reminded to change file `F2` after one or two days by developer D1, whenever D1 changed file `F1`, as shown in Figure 1. Previous work *e.g.*, [2], [3], [10], does not consider co-changed files, if they were changed by two different authors in the same period. Thus, we present, the first approach , to the best of our knowledge, to detect and to report co-changed files maintained by different developers.

Thus, we introduce the novel concepts of *macro co-changes* (MCC) and *dephase macro co-changes* (DMCC), inspired from co-changes and using the concept of change periods, as defined in Section II. The MCC describe a set of files that always change together in the same periods of time. The DMCC describes a set of files that always change together with some shift in time in their periods of change. We also consider approximate MCC and DMCC when co-changes occur "almost" all the time, using the Hamming distance and by dividing the MCC (respectively, DMCC) set into two sets: the S$_{MCC}$ (respectively, S$_{DMCC}$) set that contains files that have exactly the same (dephase) profile and the S$_{MCCH}$ (respectively, S$_{DMCCH}$) set that contains approximate (dephase) macro co-changing files.

We also present Macocha, an approach to identify MCC and DMCC in the evolution of programs, which relates to file stability and co-changes, and perform two types of empirical studies. *Quantitatively*, we compare the stability analysis of Macocha with that of UMLDiff [11] and the co-change analysis of Macocha with association rules [2], [3]. We compare the results of the three approaches on four different programs: ArgoUML, FreeBSD, SIP, and XalanC, developed with three different programming languages, C, C++, and Java. *Qualitatively*, we use external information provided by bugs reports, mailing lists, and requirement descriptions to validate the MCC and DMCC not found using association rules and to show that these MCC and DMCC explain real evolution phenomena.

Thus, the contributions of this paper are: (1) the definitions of the two novel concepts of MCC and DMCC; (2) an efficient approach to identify such co-changes, not identified by previous work, in large programs ; (3) empirical studies showing the existence and usefulness of (approximate) MCC and DMCC among files. Section II presents Macocha. Section III describes our empirical study while Section IV reports and discusses its results as well as threats to its validity. Section VI discusses related work. Section VII concludes with future work.

II. Our Approach: Macocha

We propose Macocha to mine version-control systems (CVS and SVN), 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 have similar co-changes pattern, i.e, are macro co-changing or dephase macro co-changing. We now present the concepts of our approach using examples from ArgoUML.

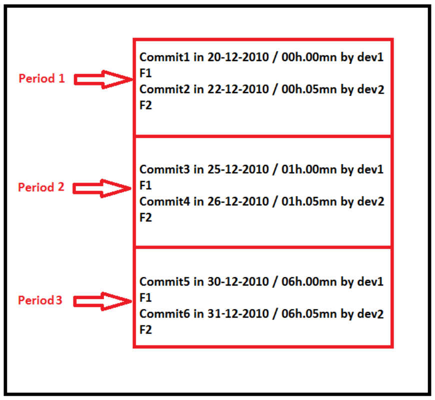

Period 1
Commit1 in 20-12-2010 / 00h.00mn by dev1
F1
Commit2 in 22-12-2010 / 00h.05mn by dev2
F2
Period 2
Commit3 in 25-12-2010 / 01h.00mn by dev1
F1
Commit4 in 26-12-2010 / 01h.05mn by dev2
F2
Period3
Commit5 in 30-12-2010 / 06h.00mn by dev1
F1
Commit6 in 31-12-2010 / 06h.05mn by dev2
F2

Figure 1. Files `F1` and `F2` are changed by different developers and in two consecutive periods of time.

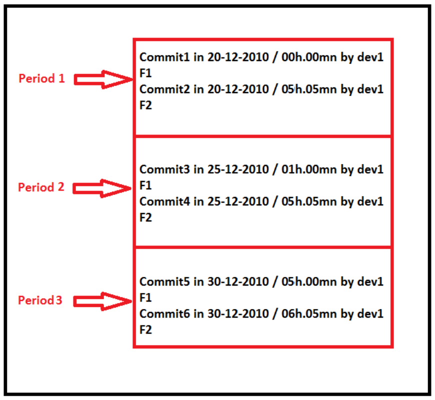

Period 1
Commit1 in 20-12-2010 / 00h.00mn by dev1
F1
Commit2 in 20-12-2010 / 05h.05mn by dev1
F2
Period 2
Commit3 in 25-12-2010 / 01h.00mn by dev1
F1
Commit4 in 25-12-2010 / 05h.05mn by dev1
F2
Period3
Commit5 in 30-12-2010 / 05h.00mn by dev1
F1
Commit6 in 30-12-2010 / 06h.05mn by dev1
F2

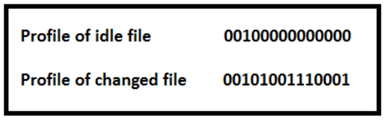
Figure 2. Two changes performed by one developer are sequential in time (after few hours), `F1` and `F2` are macro co-changing

A. Definitions

1) Change Period: We draw inspiration and extend the classical sliding window approach to consider that two subsequent changes by any author and with any log message are part of one change period if we do not detect an interrupt between these two subsequent changes. We define an interrupt as a continuous duration without a change.

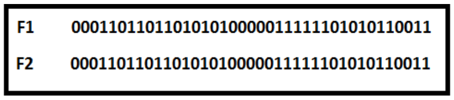
Hatton [12] presented an empirical study to estimate the time for a particular maintenance requests also known as change requests or CRs). The author showed that the average duration of a CR is 5.17 hours and that the largest duration was less than 40 hours.

Thus, in Macocha, the largest duration of a change period is less than 40 hours. Let t be the number of hours of an interrupt between two change periods. In Macocha, we choose $t = 5.17$ hours, *e.g.*, if the interrupt between two subsequent changes is more than $t = 5.17$ hours, we assume that theses two subsequent changes belong to two different change periods.



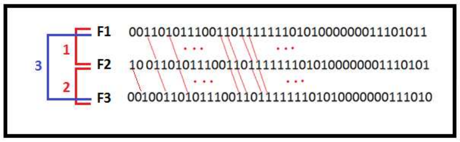
Profile of idle file	00100000000000
Profile of changed file	00101001110001

Figure 3. Profiles showing file Stability



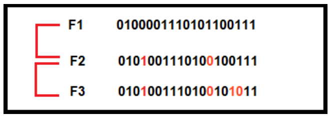
F1	00011011011010101000001111110101011001 1
F2	0001101101101010100000111111010101 10011

Figure 4. Files F1 and F2 were in macro co-change



F1 00110101110011011111110101000000011101011
F2 10 011010111001101111111010100000001110101
F3 001001101011100110111111101010000000111010

Figure 5. Three different bit vectors showing dephase macro co-change



F1 010000111010110011 1
F2 0101001110100100111
F3 0101001110100101011

Figure 6. Three different bit vectors showing approximate macro co-change

In ArgoUML: We find 2,843 change periods in 11 years of maintenance. By comparing the result of Macocha on co-change with association rules [2], [3] we find that $t = 5.17$ hours is a good trade-off between precision and recall (see Section IV for details).

2) Profile: We define a profile as a bit vector that describes if a file 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. We indicate that a file has changed in the i^{th} period by putting the i^{th} bit to one; zero otherwise.

3) File Stability: Macocha groups files according to their stability: idle and changed, as shown in Figure 3. Each group is a set of profiles with similar stability. *Idle files* do not change in any change period after their introduction into the system, *i.e.*, their profiles mostly contain zeroes, while *changed files* are files that changed after their introduction into program. Macocha use this group to identify files having similar co-changes pattern.

In ArgoUML: Macocha identifies 202 idle files and 2,946 changed files.

4) Macro Co-changes: Similar changed profiles grouped together represent MCCs and DMCCs. A S_{MCC} is two or more changed files that change together, *i.e.*, that have identical profiles during the life cycle of a program, as illustrated in Figure 4. Given a file F1, a S_{DMCC} is the set composed of F1 and one or more files, F2...FM, such that F2...FM always macro co-change with the same shift in time $s \in [0, n-1]$ with respect to F1 during the evolution of a program. Figure 5 illustrate that F1 and F2 are in dephase macro co-change with $s = 1$; F2 and F3 are in a DMCC with $s = 2$; and, F1 and F3 are in a DMCC with $s = 3$. In this paper, we limit our study to $s = 1$.

Macocha considers both identical and *similar* profiles (with or without shifts in time) to account for cases where the files did not change exactly at the same times (in terms of change periods). We use the Hamming distance D_H to measure the amount of differences between two change profiles, *i.e.*, the number of positions at which the corresponding bits are different. After analysing several values of D_H between two profiles in different system, we found that D_H<3 is the best trade-off between precision and recall (as shown in Figure 9) . Thus, in this paper, we consider that two profile are similar if the Hamming distance between them is less then three (D_H<3). Figure 6 illustrate that F1 and F2 are in approximate macro co-change with D_H<3; F2 and F3 are in a approximate MCC with D_H<3; and, F1 and F3 are in a approximate MCC with D_H<5.

B. Data Model, Implementation, and Outputs

Figure 7 describes the data used by Macocha. A change contains several attributes: the changed file names, the dates of changes, the developers having committed the changes. Using this data, Figure 8 illustrates the concrete process of Macocha. Macocha takes as input a CVS/SVN change log. It creates a profile that describes the evolution of each file in each change period. It uses these profiles to compute the stability of the files and, then, identify MCC and DMCC.

Macocha returns the following sets of (dephase) macro co-changing files (and their profiles): S_{MCC}, the set of macro co-changing files with identical profiles in a program and S_{DMCC}, the set of dephase macro co-changing files identified when shifting profiles by s change periods. S_{MCCH}, the set of approximate macro co-changing files with similar profiles in a program by using the Hamming distance (0<D_H<3) and S_{DMCCH}, the set of approximate dephase macro co-changing files identified when shifting profiles by s change periods.

III. Empirical study

Following GQM [13], the goal of our study is to show that our approach can identify MCC and DMMC and that they describe interesting evolution phenomena. Our purpose is to bring generalisable, quantitative evidence on the existence of MCC and DMCC. The quality focus is that changing one file may impact the files that (dephase) macro co-change with it. 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 both the comprehension and the maintenance of programs.

A. Research Questions

We formulate two research questions: **RQ1:** How does Macocha compare to previous work in term of precision

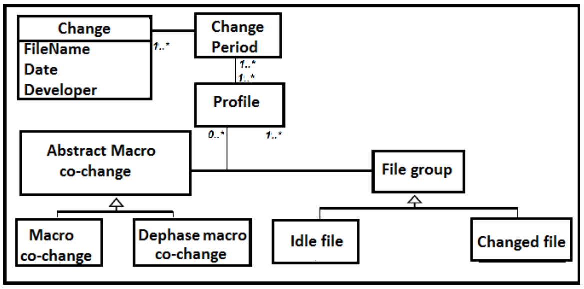


Figure 7. Meta-model of our data

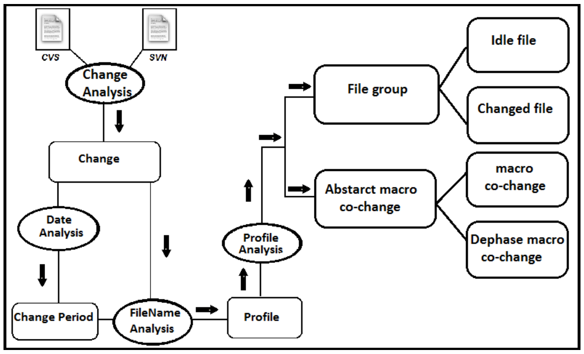


Figure 8. Analysis-process

and recall? **RQ2:** Are there (approximate) dephase macro co-changes among files and what is their usefulness?

B. *Analyses*

To answer RQ1 and RQ2, we apply Macocha to different object programs and collect the different sets of MCC and DMCC. We then perform two types of empirical studies.

Quantitatively, on the one hand, we compare the results of Macocha with those of UMLDiff for file stability. We thus 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 our needs. Idles files do not change in any change period after their introduction into the program. Thus, we discard this group of files because they are not useful for the co-change analysis due to their rare evolution.

On the other hand, we compare the results of Macocha with those of the association rules approach [3] for co-changing files. We also thus show that the set S_{MCC} produced by Macocha includes the same co-changing files as reported using association rules plus new co-changing files.

Qualitatively, we confirm that each MCC found by Macocha but not association rules approach [3] is indeed a dependency link using external information from bug-reports, requirement descriptions, and mailing lists. We also select typical examples of MCCs and DMCCs and show their usefulness using external sources of information.

We thus report a quantitative analysis in accordance with the state of the art and a qualitative analysis in accordance with external information. We also report and discuss the cardinalities of the MCC and $DMCC$ sets.

We do not report performance because, using a standard computer with a Intel Core i7-740QM (1.73/ 2.93GHz), 6GB RAM, and 1GB VRAM, Macocha identifies (dephase) macro co-changes in FreeBSD (the largest program in terms of number of files and of changes) in less than ten minutes.

C. *Objects*

We choose four programs developed with three different programming languages: ArgoUML[4], FreeBSD[4], SIP[4], and XalanC[4]. We use these programs because they are open source, have been used in previous work [10] [14], are of different domains and in different programming languages, span several years and versions, and underwent between thousands and hundreds of thousands of changes. Table I summarises some programs statistics.

ArgoUML 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 11 years, from 1998-01-26 to 2009-01-29. In this period, ArgoUML has gone through over 13 major versions, from the first published version to version 0.26.2 in November 2008, and many more minor versions.

FreeBSD 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 15 years, from 1994-05-25 to 2009-02-11. In this period, FreeBSD has gone through eight major versions, from the first published version to version 7.0 on February 2008.

SIP Communicator 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 for a period of five years, from 2005-07-21 to 2010-12-09.

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 11 years, from 1999-12-18 to 2009-01-17. In this period, XalanC has gone through over 20 major versions, from the first published version to version 1.10 in November 2008.

IV. Study Results and Discussions

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

A. *Data Preprocessing and Identifying changed files*

Before finding patterns in the change history, Macocha detects in each program the set of changed files. The pre-processing step involves eliminating idles files because they

[4] http://argouml.tigris.org/, http://www.freebsd.org/, http://www.sip-communicator.org, and http://xml.apache.org/xalan-c/

	ArgoUML	FreeBSD	SIP	XalanC
Languages	Java	C	Java	C++
Versions	30	8	2	21
Files	3,148	3,603	2,790	529
Changes	16,727	186,959	8,046	397,052
Start Dates	98-01-26	94-05-25	05-07-21	99-12-18
End Dates	09-01-29	09-02-11	10-12-09	09-01-17
CPs	2,843	1,121	1,553	924

Table I
DESCRIPTIVE STATISTICS OF THE OBJECT PROGRAMS (CPS: NUMBERS OF CHANGE PERIODS)

	ArgoUML	FreeBSD	SIP	XalanC
Idle files	202	1,856	963	7
Changed files	2,946	1,747	1,827	522
# of S$_{MCC}$	166	121	142	36
Max # files	35	24	15	17
Min # files	2	2	2	2
# of S$_{MCCH}$	196	163	182	85
Max # files	46	44	32	22
Min # files	2	2	2	2
# of S$_{DMCC}$	11	1	6	1
Max # files	4	2	3	2
Min # files	2	2	2	2
# of S$_{MCCH}$	53	63	36	4
Max # files	6	8	5	2
Min # files	2	2	2	2

Table II
CARDINALITIES OF THE SETS OBTAINED IN THE EMPIRICAL STUDY

		Idle Groups	Changed Groups
ArgoUML	Idle Clusters	202	0
	Short-lived Clusters	0	1,390
	Active Clusters	0	1,556
SIP	Idle Clusters	963	0
	Short-lived Clusters	0	997
	Active Clusters	0	830
XalanC	Idle Clusters	7	0
	Short-lived Clusters	0	291
	Active Clusters	0	231

Table III
CARDINALITY OF MACOCHA SETS IN COMPARISON TO UMLDIFF [11]

do not change in any change period after their introduction into the program. Thus, they do not participate in co-change patterns.

Table III reports the number of idle, short-lived, and active files found by UMLDiff in the object-oriented object programs (ArgoUML, SIP, and XalanC) and their categorisation by Macocha. Because we want to distinguish idle from changed files, Macocha groups together the files identified as short-lived and active by UMLDiff and compare the sets provided by UMLDiff and by Macocha and find that they are identical. For example, Macocha finds 2,946 changed files in ArgoUML, identical to the UMLDiff $1,390 + 1,556 = 2,946$ short-lived and active files.

In addition, Macocha computes file stability in few minutes because, unlike UMLDiff, which takes few hours [15], because it does not create UML-like representations of the programs before performing its analysis. Macocha can analyse file stability for any program, unlike UMLDiff, providing that CVS/SVN repositories are available.

In ArgoUML: We detect 202 idle files. For example, the files `ModeChangeEvent.java` and `GoModelToClassifiers.java` were modified in only one change period in 11 years.Using UMLDiff, we confirm that these files belong to an idle cluster.

We detect 2,946 changed files. For example, the files `TestProject.java` and `NotationUtilityUml.java` were modified 20 times during the evolution of ArgoUML. Thus, these files belong to the changed group. Using UMLDiff, we confirm that these files belong to an active cluster.

In FreeBSD: We find 1,856 idle files. For example, `hdtimer.c` and `hddebug.c` were modified in one change period in 15 years.

We detect 1,747 changed files. The files `subrclist.c` and `stallion.c` were modified in 15 change periods during the evolution of FreeBSD. We cannot use UMLDiff to verify this result because UMLDiff can not analyse programs written in C.

In SIP: We obtain 963 idle files. For example, `SelectImagePanel.java` and `ImageSourceStream.java` were modified in one change period in five years. With UMLDiff, these files belong to an idle cluster. Macocha detect 1,827 changed files. The files `DefaultTreeContactList.java` and `TreeContactList.java` were modified 15 times during the evolution of the program.

In XalanC: Our approach detect seven idle files. For example, `XLocator.cpp` and `Cloneable.cpp` were modified in one change period in nine years. Using UMLDiff, we confirm that these files belong to an idle cluster. Macocha detect 522 changed files. The files `TopLevelArg.cpp` and `XalanEXSLTSet.cpp` were modified in 30 changes periods during the evolution of the program. Using UMLDiff, we confirm that these files belong to an active cluster.

B. How does Macocha compare to previous work in term of precision and recall?

For each program, Macocha detect files that have identical or similar profiles (the MCCs sets) and report them.

Quantitatively: We compare the S$_{MCC}$ found by Macocha with the co-changing files found by an approach based on association rules [3] (see also [10]), which uses the Apriori algorithm [16] 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 [3] and a minimum support of 2 to compare association rules and our approach.

We thus perform an *internal evaluation* similar to that of Zimmermann *et al.*'s. Given snapshots S_i, $i \in [1, ..., n]$, we build two equal sets T_{train} = $\{S_1...S_t\}$ and T_{test} =

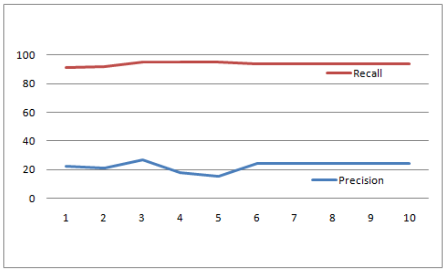


Figure 9. The mean of Precision and Recall achieved by Macocha with different values of D$_H$ for the four programs

	Association Rules		Macocha	
	Precision	Recall	Precision	Recall
ArgoUML	15%	66%	20%	75%
FreeBSD	22%	100%	24%	100%
SIP	18%	89%	24%	91%
XalanC	16%	100%	22%	100%

Table IV
ASSOCIATION RULES'S APPROACH [3] VS. MACOCHA

$\{S_{t+1}...S_n\}$. We use T_{train} to build association rules and compare the co-changing files in T_{train} with those in T_{test}.

For the four programs, we find that Macocha improve precision and recall over the approach based on association rules, as shown in Table IV. For example, for ArgoUML, results indicate that, the precision and the recall of Macocha, respectively 20% and 75%, are better than those of association rules, respectively 15% and 66%.

The rationale of an internal evaluation is that no expert 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 have some limits [17] [18]: (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 positive; (3) If the training set contained false positives or negatives, they cannot be detected using the testing set.

Qualitatively: To overcome these limits and to validate the S$_{MCC}$ not found using association rules, we also perform an *external evaluation* of Macocha by considering the results of the association rules as an oracle and by manually comparing them with those of Macocha. In fact, for each set returned by Macocha, if an identical set is returned by association rules, it is considered a true positive. If the two sets are not identical, we use external information to validate missing files and to decide if they present a true positive, a false negative, or a false positive. For example, In XalanC, all the sets detected by association rules are detected by Macocha except three sets. We validated these three sets using two messages in the mailing list and one bug in the Bugzilla.

	Association Rules		External Information	
	Precision	Recall	Precision	Recall
ArgoUML	86%	98%	100%	99%
FreeBSD	98%	100%	100%	100%
SIP	85%	96%	100%	98%
XalanC	90%	100%	100%	100%

Table V
EXTERNAL EVALUATION OF MACOCHA WHEN USING THE RESULTS OF ASSOCIATION RULES *et al.*'S APPROACH [3] AS ORACLE AND AFTER MANUAL VALIDATION USING EXTERNAL INFORMATION

Table V reports, under the External Information header, the precision and recall values of Macocha after manual validation, which show that Macocha is able to detect S$_{MCC}$ missed and co-changes wrongly reported by association rules. We do not obtain 100% recall because of our choice of $t = 5.17$ hours. A smaller value of t would yield a higher recall but a lower precision. We thus confirm Hatton's study [12] and report that $t = 5.17$ hours is a good trade-off between precision and recall.

Table V also reports, under the Association Rules header, the precision and recall of Macocha with respect to the approach based on association rules. It shows that Macocha detects the majority of co-changing files detected by association rules in the four object programs. In addition, Macocha detects other S$_{MCC}$ not detected by association rules. For example:

In ArgoUML: `ClassifierRole.java` and `MessageDirectionKind.java` were in approximate MCC. In fact, in the Bugzilla of ArgoUML, the bug ID 881[5] states that "when classifier assigned to an object [...] ArgoUML stops responding" in relation with these two files. These two file were changed by different authors in a time-window of more than few minutes. Thus, by applying the association rule approach described in [3], we cannot find that these files are co-changing.

In SIP: `StatusIcon.java` and `ContactPanel.java` were in MCC. As confirmed in the Bugzilla of SIP by the bug ID 497[6]. This bug describes on an "Status notifications for a choosen contact" relating these two files. These two files were changed by the same developer `yanas` five times in a time-window of more than few minutes. Thus, by applying the association rule approach described in [3], we could not find that these files are co-changing.

C. Are there (approximate) dephase macro co-changes among files and what is their usefulness?

To the best of our knowledge, no previous approach can detect files maintained with similar trends and some given shifts in time. We validate the usefulness of DMCCs using external information. For the lack of space, we cannot illustrate all DMCCs, so we only report representative examples.

In ArgoUML: When developers changed `ActionAlign.java`, developers changed `ForwardingComponent.java` in the following change period. Thus, these two files

[5] http://argouml.tigris.org/issues/show_bug.cgi?id=881
[6] http://java.net/jira/browse/JITSI-134

are in DMCC. In fact, in the Bugzilla of ArgoUML, the bug ID 1957[7] relates the two files: “Each label text is a few pixels too high for its component. They should be positioned such that the label text is vertically aligned with the text in the labeled component”.

In FreeBSD: We find that `ah-core.c` and `hpfs-alsubr.c` are in approximate DMCC. In the mailing list of FreeBSD, the `Message-ID: <200906011106.n51B62Da020139@freefall-.freebsd.org>` states that the two files are related in a lengthy the message from bugmaster@FreeBSD.org on June 1, 2009 about “Current problem reports”.

In SIP: We find that `MuteDataSource.java` and `CallPeerActionMenuBar.java` were changed systemically with one shift change period in five years. In fact, These two files implement the same feature[8]: Audio-Calls.

In XalanC: We find that `Cloneable.cpp` and `XLocator.cpp` are in approximate DMCC. In the XSLT syntax and semantic specification[9], these files are related: “A single template [...] can pull string values out of arbitrary locations in the source tree; it can generate structures that are repeated according to the occurrence of elements”.

In the following scenarios, we summarise the usefulness of DMCCs reported by Macocha.

1) Management of Development Teams: If two classes are in (approximate) dephase macro co-change, they should ideally be maintained by the same team of developers to minimise the risks of introducing bugs in the future. The team of developers most likely possesses a wealth of unwritten knowledge about the design and implementation choices that they made for these classes, which would help them to prevent introducing bugs [19].

Consequently, a team leader should redefine the organisation of the maintenance team according to the DMCCs 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[10] [11] [12] dephase macro co-changing files that have generated bugs, we found that these changes have been made with one shift in time in their periods of change and by different developers . Thus, such co-changes can not be detected by previous work. Thanks to DMCCs, a team leader should ensure that team who will maintain these files in each change period have the necessary knowledge to maintain the dependency among these files.

2) Bug and Change Propagation: Knowing that two files are in DMCCs implies the existence of (hidden) dependencies between these two files. If these dependencies are not properly maintained, they can introduce bugs in a program. 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 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 [3], we cannot find that these files are co-changing. Thus, by knowing files that are in DMMCs, we could explain and possibly prevent bugs; we plan to study in future work the bug prediction using (approximate) dephase macro co-changes.

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. The version history may reveal hidden links that relate files and would be sufficient to attract the developers’ attention. For example, in SIP, we detect traceability links between four approximate dephase macro co-changing files. By applying the association rule approach described in [3], we cannot 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. 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. Thus, by knowing classes there are in (approximate) dephase macro co-change, we could detect potentially traceability links between them, which we plan to concretely study in future work.

V. Discussions

With our approach, we detect files in MCCs or in DMCCs in four different programs belonging to different domains and with different sizes, histories, and programming languages. However, we do not detect MCCs and DMCCs with the same proportion in each program. We observe that the numbers of MCCs and DMCCs found in the programs developed in Java (ArgoUML and SIP) are greater than the number of MCCs and DMCCs found in program developed in C or C++ (see Table II). We explain this finding by the fact that, on the one hand, the majority of FreeBSD files are idle and that, on the other hand, XalanC is the smallest program analysed. Thus, we also apply our approach to detect (dephase) macro co-changes on fewer C and C++ files than Java files, less than 529 files, thus explaining the lower numbers of MCCs and DMCCs. In future work, we will conduct studies on other programs in these languages to confirm this observation and to assess the numbers of MCCs and DMCCs according to the programming languages.

A. Threats to the Study Validity

Some threats limit the validity of our empirical study.

[7] http://argouml.tigris.org/issues/show_bug.cgi?id=1957

[8] http://www.jitsi.org/index.php/Main/Features

[9] http://www.w3.org/TR/xslt

[10] http://argouml.tigris.org/issues/show_bug.cgi?id=1957

[11] http://argouml.tigris.org/issues/show_bug.cgi?id=2926

[12] http://argouml.tigris.org/issues/show_bug.cgi?id=4604

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 changes in 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, MCCs and DMCCs exist and are corroborated by external sources of information (bug reports and others). **Actually, we apply static analysis to detect MCCs and DMCCs because co-change analysis is known to be more useful when combined with static analysis [23].** 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 four 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 manipulate 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. The change logs and the changed files of the four programs analysed with their profiles to obtain our observations are on-line at http://www.ptidej.net/downloads/experiments/wcre11b/.

External Validity: We performed our study on four different real programs belonging to different domains and with different sizes, histories, 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; future work includes replicating our study in other contexts and with other programs.

VI. Related Work

The concepts of MCCs and DMCCs relate our work to that on file stability, co-change, and change propagation.

A. File Stability

Many approaches exist to group files based on their relative stability throughout the software development life cycle. For example, Kpodjedo *et al.* [20] 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 Mozilla class diagram by collecting 144 Mozilla snapshots over six 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, thus, that likely constitute the stable backbone of Mozilla.

As other example, UMLDiff [11] 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 heuristic-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).

The Error Tolerant Graph Matching algorithm and UMLDiff take few hours to analyse file stability for the four programs analysed in this paper because they require parsing and comparing AST-like representations of the programs before performing their analyses. Macocha computes stability in few minutes using the change periods of a program, which depend on how the developers of the program organise their work and group changes through the life cycle of the program.

B. Co-changing Files

Ying *et al.* [2] and Zimmermann *et al.* [3] 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 algorithm uses co-change history in CVS and avoids the source code dependency parsing process. However, it only computes the frequency of co-changed files in the past and omits many other cases, *e.g.*, files that co-change with always the same period of time between changes. In Section IV, we showed that approaches based on association rules cannot detect all occurrences of MCCs and any occurrences of DMCCs 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.

German [7] 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. Thus, Softchange can not detect co-changed file maintained in the same time by different developers. Ceccarelli *et al.* [21] and Canfora *et al.* [10] 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 to forecasting the 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. If the authors integrate the analysis of files that are maintained by different developers in periods of time of more than few minutes, their approach could then detect typical examples of MCCs and DMCCs.

Antoniol *et al.* [8] presented an approach to detect similarities in evolutions of files starting from past maintenance, notwithstanding their temporal distortions. 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 can find files having very similar maintenance evolution history but they did not present a tool to detect MCCs and DMCCs. It used cepstral distance to assess series similarity (if two cepstra series are "close", the original signals have a similar evolution in time) with which we can not distinguish between the occurrences of MCCs and DMCCs.

C. *Change Propagation*

The development and maintenance of a program involves handling a large number 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. Change propagation analyses how changes made to one file propagate to others. Law and Rothermel [22] 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 control flow, including loops iteration and inter-procedural paths. Law *et al.*'s approach builds a representation of a program 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 [23] 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.* [24] 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 [25] 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.

Theses approaches detect change propagation among files. Their change-propagation model can be used to predict future change couplings and may involve several files that are in MCCs or in DMCCs but they do not allow to differentiate between these two concepts. All these approaches grouped change couplings created by the same author and have the same log message; thus, they can not detect (approximate) MCCs and–or DMCCs.

Ambros *et al.* [18] 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. Beyer and Hassan [26] introduced the evolution storyboard, a new concept for animated visualisations of historical information about the program structure, and the storyboard 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.

However, Xing and Stroulia [27], reported that these visualisations are limited in their applicability because they assume a substantial interpretation effort of their users and they do not scale well: they become unreadable for large systems with numerous components.

VII. Conclusion and Future Work

We introduced the novel concepts of macro co-changes and dephase macro co-changes to describe that two files were changed by developers within same change periods, with possible shifts in time. We describe, Macocha, an approach to detect (dephase) macro co-changes using file profiles and their stability in time.

Macocha relates to file stability and co-changes. We therefore performed two types of empirical studies. Quantitatively, we compared Macocha with UMLDiff [11] and an association rules approach [3] by applying and comparing the results of the three approaches on four different programs:

ArgoUML, FreeBSD, SIP, and XalanC, and showed that Macocha can identify the same idle/changed files as UMLDiff and that Macocha has a better precision and recall than the approach based on association rules. Qualitatively, we used external information provided by bugs reports, mailing lists, and requirement descriptions to show that detected MCCs and DMCCs explain real, important evolution phenomena. We also showed that dephase macro co-changes do exist and can help in explaining bugs, managing development teams, and traceability analysis.

We are currently (1) replicating our studies with other programs, (2) performing a comprehensive study of the number of MCCs and DMCCs with varying values of t and s (especially dependent on the analysed programs), (3) identifying other scenarios in which dephase macro co-changes help, and (4) relating MCCs and DMCCs with static analysis and external software characteristics, such as change proneness. Future work also includes a comparative study of the different sets computed by Macocha and associations rules with different value of confidence and support other than the values reported in [3].

ACKNOWLEDGMENT

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

REFERENCES

[1] M. M. Lehman and L. Belady, Eds., *Program evolution: processes of software change*. Academic Press Professional, Inc., 1985.

[2] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, “Predicting source code changes by mining change history,” *Transactions on Software Engineering*. IEEE Computer Society Press, 2004, vol. 30, no. 9, pp. 574–586.

[3] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, “Mining version histories to guide software changes,” in *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, 2004, pp. 563–572.

[4] H. Gall, K. Hajek, and M. Jazayeri, “Detection of logical coupling based on product release history,” in *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 1998, pp. 190–.

[5] A. Mockus, R. T. Fielding, and J. D. Herbsleb, “Two case studies of open source software development: Apache and mozilla,” *ACM Trans. Softw. Eng. Methodol.* ACM, July 2002, vol. 11, pp. 309–346.

[6] M. Fischer, M. Pinzger, and H. Gall, “Populating a release history database from version control and bug tracking systems,” in *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 2003, pp. 23–.

[7] D. M. German, “An empirical study of fine-grained software modifications,” *Empirical Softw. Engg.* Kluwer Academic Publishers, September 2006, vol. 11.

[8] G. Antoniol, V. F. Rollo, and G. Venturi, “Linear predictive coding and cepstrum coefficients for mining time variant information from software repositories,” in *Proceedings of the International Workshop on Mining software repositories*. ACM Press, 2005, pp. 1–5.

[9] S. Bouktif, Y.-G. Guéhéneuc, and G. Antoniol, “Extracting change-patterns from cvs repositories,” in *Proceedings of the 13th Working Conference on Reverse Engineering*. IEEE Computer Society, 2006, pp. 221–230.

[10] G. Canfora, M. Ceccarelli, L. Cerulo, and M. Di Penta, “Using multivariate time series and association rules to detect logical change coupling: An empirical study,” in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*. IEEE Computer Society Press, pp. 1–10.

[11] Z. Xing and E. Stroulia, “Analyzing the evolutionary history of the logical design of object-oriented software,” *Transactions on Software Engineering*. IEEE Computer Society Press, 2005, vol. 31, pp. 850–868.

[12] L. Hatton, “How accurately do engineers predict software maintenance tasks?” *Computer*. IEEE Computer Society Press, 2007, vol. 40.

[13] V. R. Basili and D. M. Weiss, “A methodology for collecting valid software engineering data,” *Software*. IEEE Computer Society Press, 1984, vol. 10, no. 6, pp. 728–738.

[14] T. Zimmermann, S. Breu, C. Lindig, and B. Livshits, “Mining additions of method calls in argouml,” in *Proceedings of the International Workshop on Mining Software Repositories*. ACM Press, 2006.

[15] Z. Xing and E. Stroulia, “Umldiff: an algorithm for object-oriented design differencing,” in *Proceedings of the 20th International Conference on Automated Software Engineering*. ACM Press, 2005.

[16] R. Agrawal and R. Srikant, “Fast algorithms for mining association rules in large databases,” in *Proceedings of the 20th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 1994.

[17] A. Vanya, S. Klusener, N. van Rooijen, and H. van Vliet, “Characterizing evolutionary clusters,” in *Proceedings of the 16th Working Conference on Reverse Engineering*. IEEE Computer Society, 2009.

[18] M. D’Ambros, M. Lanza, and M. Lungu, “Visualizing co-change information with the evolution radar,” *Transactions on Software Engineering*. IEEE Computer Society Press, 2009, vol. 35, no. 5, pp. 720–735.

[19] B. W. Rebecca Wirfs-Brock and L. Wiener, Eds., *Designing Object-Oriented Software*. Prentice Hall, 1990.

[20] S. Kpodjedo, F. Ricca, P. Galinier, and G. Antoniol, “Recovering the evolution stable part using an ecgm algorithm: Is there a tunnel in mozilla?” in *CSMR*, 2009, pp. 179–188.

[21] M. Ceccarelli, L. Cerulo, G. Canfora, and M. Di Penta, “An eclectic approach for change impact analysis,” in *Proceedings of the 32nd International Conference on Software Engineering*. ACM Press, 2010, pp. 163–166.

[22] J. Law and G. Rothermel, “Whole program path-based dynamic impact analysis,” in *Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society, 2003, pp. 308–318.

[23] A. E. Hassan and R. C. Holt, “Predicting change propagation in software systems,” in *Proceedings of the 20th IEEE International Conference on Software Maintenance*. IEEE Computer Society, 2004, pp. 284–293.

[24] Y. Zhou, M. Würsch, E. Giger, H. C. Gall, and J. Lü, “A bayesian network based approach for change coupling prediction,” in *Proceedings of the 15th Working Conference on Reverse Engineering*. IEEE Computer Society, 2008, pp. 27–36.

[25] G. Canfora and L. Cerulo, “Impact analysis by mining software and change request repositories,” in *Proceedings of the 11th IEEE International Software Metrics Symposium*. IEEE Computer Society Press, 2005, p. 29.

[26] D. Beyer and A. E. Hassan, “Animated visualization of software history using evolution storyboards,” in *Proceedings of the* $13th$ *Working Conference on Reverse Engineering*. IEEE Computer Society Press, 2006.

[27] Z. Xing and E. Stroulia, “Bottom-up design evolution concern discovery and analysis,” Tech. Rep., 2007.