Research

Detecting Asynchrony and Dephase Change Patterns by Mining Software Repositories

Fehmi Jaafar[a,b], Yann-Gaël Guéhéneuc[a], Sylvie Hamel[b], and Giuliano Antoniol[c]

[a] PTIDEJ *Team, DGIGL, École Polytechnique de Montréal, Québec, Canada*
[b]*LBIT Lab., DIRO, Université de Montréal, Québec, Canada*
[c] *SOCCER Lab., DGIGL, École Polytechnique de Montréal, Québec, Canada*

SUMMARY

Software maintenance amounts for the largest part of the costs of any program. To reduce this part of the costs, previous work proposed approaches to identify the artefacts of programs that change together and, thus, reveal the (hidden) dependencies among changing artefacts. These approaches analyse historical data, mined from version control systems, and report change patterns, which hint at the causes, consequences, and actors of the changes to software artifacts, generally 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, akin to *macro co-changes* (MCC), *i.e.*, of files that co-change within a large time interval (change periods), and the Dephase change pattern, akin to *dephase macro co-changes* (DMCC), *i.e.*, macro co-changes that always happen with the same shifts in time. We use the k-nearest neighbor algorithm to group changes into change periods and we use the Hamming distance to detect approximate occurrences of MCC and DMCC. We present our approach, Macocha, to identify these two change patterns in large programs. We apply Macocha and compare its results in terms of precision and recall with UMLDiff (file stability) and association rules (co-changing files) on seven systems: ArgoUML, FreeBSD, JFreeChart, Openser, SIP, XalanC, and XercesC, developed with three different languages (C, C++, and Java) and of sizes ranging from 532 to 1,693 C, C++, or Java files and from 1,555 to 23,944 change commits. We also use external information and static analyses to validate the (approximate) MCC and DMCC found by Macocha. We thus show the existence and usefulness of theses novel change patterns to ease software maintenance and, thus potentially, reduce related costs.

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

1. Introduction

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

[1]Correspondence to: Fehmi Jaafar – Jaafarfe@iro.umontreal.ca

to the point of disappearing. Indeed, developers need knowledge to identify hidden dependencies among programs' artefacts to improve the speed and accuracy of changes while reducing maintenance's cost.

The literature describes several approaches to extract and analyse such hidden dependencies and to infer the patterns that describe these changes to help developers maintain their programs [BKZ10, KR11]. Artefacts can be source code files, classes in object-oriented programs, specifications, and so on. 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], which represent the (often implicit) dependencies or logical couplings among 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 ms. [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 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, because previous approaches cannot detect change patterns between files with long time intervals between their changes and–or performed by different developers and with different log message, they miss occurrences of other change patterns. We also showed that such change patterns provide interesting information to developers. For example, in the Bugzilla of ArgoUML, the bug ID 5378[2] 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 have a relationship, which is hidden because we cannot detect dependencies among 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 `bobtarling` but their changes were always separated by a 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.

Let `F1` and `F2` be two file 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 few minutes to group changes committed by the same developer and with the same log message. Yet, such co-change contains important information both for the developer and her colleagues: changes to `F1` must likely propagate to `F2` because these two files have a, possibly hidden, dependency, to avoid introducing bugs in the program.

As another example in ArgoUML, we found that the developers `mvw` and `tfmorris` contributed with some patches that contains `NotationUtilityJava.java` and `ModelElementName-`

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

Copyright © 2000 John Wiley & Sons, Ltd.

NotationUml.java[3] and the bug ID 2926[4] confirms that the two files have dependencies (see Section 4 for 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 mvw and tfmorris in two consecutive times: first NotationUtilityJava.java and, subsequently after some hours, ModelElementNameNotationUml.java, pointing out dependency among these files.

This previous scenario from ArgoUML happens when a developer D2 is always reminded to change file F2 after some time by developer D1, whenever D1 changed file F1, as illustrated in Figure 2. Previous work, *e.g.*, [YMNCC04], [ZWDZ04], [CCCDP], 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 to release a program with a bug because of a mismatch between files F1 and F2.

In this paper, we describe Macocha, an approach to detect two novel change patterns, summarised in our previous work [JGHA11] and detailed in Section 2. Macocha detects the Asynchrony change pattern, *macro co-changes* (MCC), and the Dephase change pattern, *dephase macro co-changes* (DMCC). 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 algorithm [Das91] to group changes into their change periods.

The Asynchrony change pattern (MCC) describes a set of files that always change together in the same change periods. The Dephase change pattern (DMCC) describes a set of files that always change together with some shift in time in their periods of changes. We also consider approximate MCC and DMCC when co-changes occur “almost” always in the same change periods, 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 follow exactly the same (dephase) change pattern and the S_{MCCH} (respectively, S_{DMCCH}) set that contains approximate (dephase) macro co-changing files.

As shown in [JGHA11], Macocha relates to file stability and co-changes. We thus perform two types of empirical studies. *Quantitatively*, we compare the stability analysis of Macocha with that of UMLDiff [XS05a] and the co-change analysis of Macocha with the state-of-the-art association rules [YMNCC04, ZWDZ04]. We also 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 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.

Thus, this paper extends our previous work [JGHA11] with the following contributions. First, we describe in more details our approach, Macocha. Second, we use the k-nearest neighbor algorithm (KNN) 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 new programs, JFreeChart, Openser and XercesC, developed with three different languages: C, C++ and

[3] http://argouml.tigris.org/issues/showattachment.cgi/2118/20101116-patch-notation.txt
[4] http://argouml.tigris.org/issues/show_bug.cgi?id=2926

Copyright © 2000 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

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 its validity. Section 6 discusses related work. Section 7 concludes with future work.

2. 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 follow the Asynchrony or Dephase change patterns, *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", *i.e.*, the dates of these commits are "close" to one another. We need the concept of change period because we analyse different programs belonging to various domains and with different sizes, histories, and programming languages: the change periods (beginning dates and durations) cannot be the same in each program.

Consequently, we want to identify the change periods in a program by grouping all the changes committed in nearby dates, independently of the developers who committed them and of their log messages. To identify changes occurring closely to one another, *i.e.*, belonging to a same change period, with use the k-nearest neighbor algorithm (KNN). The KNN is an non-parametric learning algorithm[KZP07] that does not make any assumptions on the underlying data distribution.

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. Thus, we set the initial duration of the change periods to 40 hours.

Our KNN-based algorithm to identify the change periods in a program history works as follows, as illustrated on Figure 3:

- First, we divide the whole maintenance period, from first to last considered changes, of a given program into equal sub-periods. Thus, following Hatton, 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.

Copyright © 2000 John Wiley & Sons, Ltd.

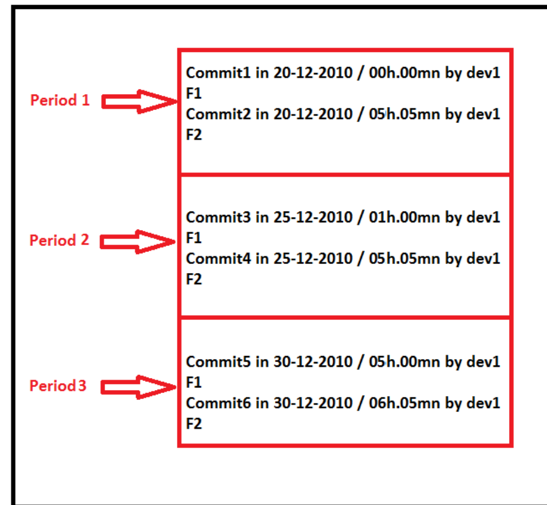


Figure 1. Two changes performed by one developer on `F1` and `F2` are sequential in time (after few hours), `F1` and `F2` follow the Asynchrony change pattern

- Third, we use the KNN with $k = 2$ 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 term of date of commit, whatever their developers and their log messages, and even if its change date is earlier than the beginning date of the change period.
- Fourth, when the KNN has assigned all changes into the different 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.

In section 4, we show that using this algorithm to group changes into change periods allow us to improve precision and recall over previous approaches.

In ArgoUML: We find 290 change periods in two years of maintenance. 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 equal to 27 hours 8 minutes 14 seconds 640 ms. and the standard deviation is equal to 12 hours 6 minutes 4 seconds 572 ms.

Copyright © 2000 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

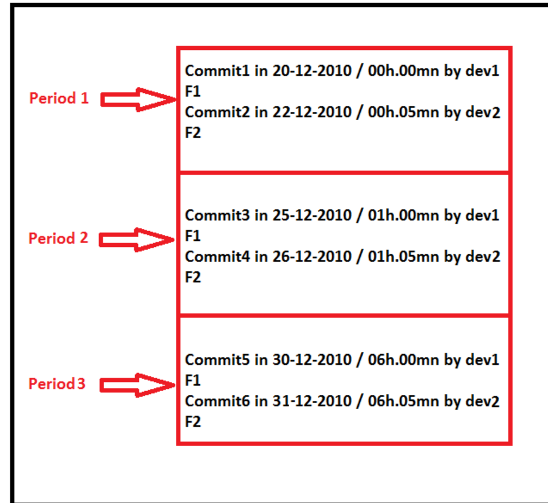


Figure 2. Files `F1` and `F2` are changed by different developers and in two consecutive periods of time, `F1` and `F2` follow the Dephase change pattern

2.1.2. *Profile*

We define a profile as a bit vector that describes if 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. We indicate that a file has changed in the ith period by putting the ith bit to one; zero otherwise.

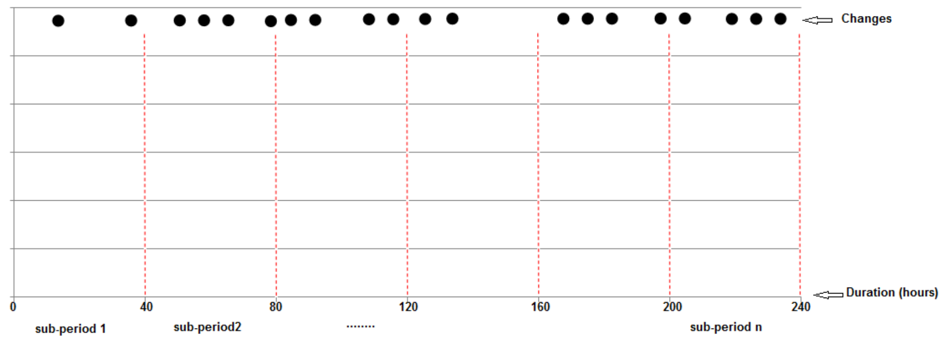
Thus, for each file in a program, its profile is defined as a vector $x = x_1...x_n$, where n represents the number of change periods. The value of x_i indicates whether the file F is changed or not at the ith change period.

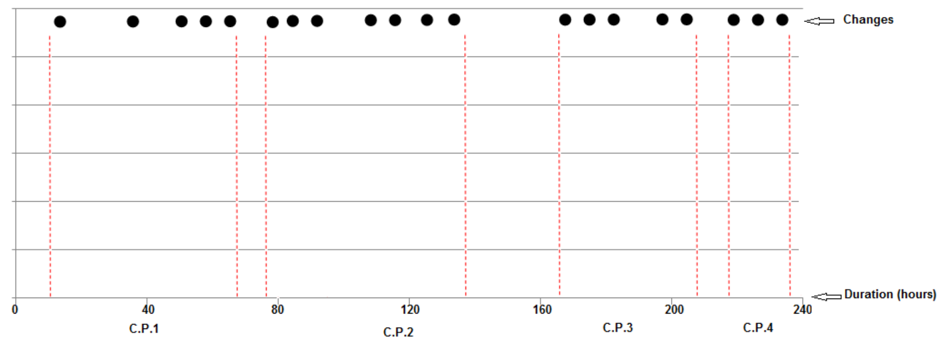
$$x_i = \begin{cases} 1 & \text{if file is changed in the change period } i \\ 0 & \text{otherwise.} \end{cases}$$

We use bit vectors because they allow efficient operations: setting a bit is constant in time, $\mathbf{O}(1)$; computing the union or intersection of two bit vectors, or the complement of a bit vector, is linear in time, $\mathbf{O}(n)$. Moreover, it is often possible to implement these operations with instruction-machine processing 32 or 64 bits simultaneously.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using smrauth.cls





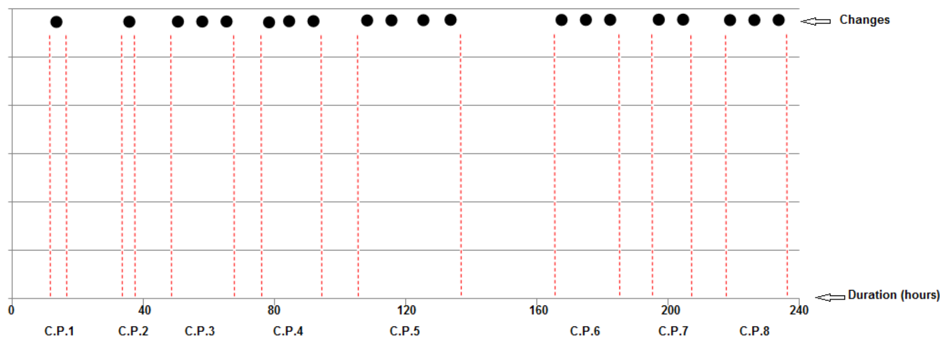


Figure 3. Illustration of the KNN-based algorithm's steps to identify the change periods in a program history

Copyright © 2000 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

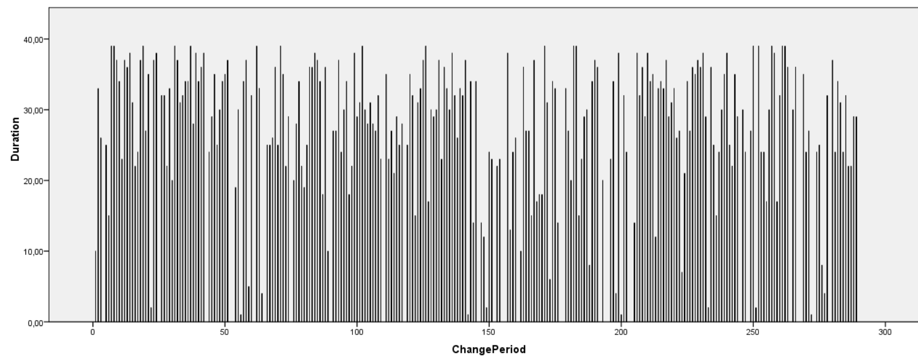


Figure 4. Change periods durations in ArgoUML detected by the KNN algorithm

Profile of idle file	**00100000000000**
Profile of changed file	**00101001110001**

Figure 5. Profiles showing file Stability

F1	**000110110110101010000011111101010110011**
F2	**000110110110101010000011111101010110011**

Figure 6. Files `F1` and `F2` follow the same occurrence of the Asynchrony change pattern

Copyright © 2000 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

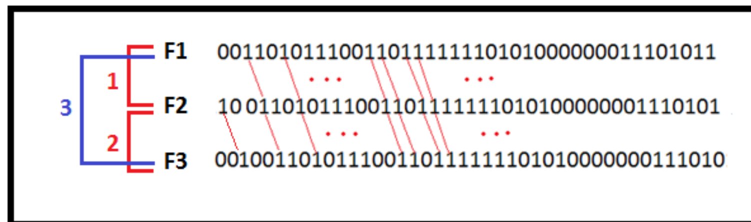


Figure 7. Three different bit vectors showing dephase macro co-change

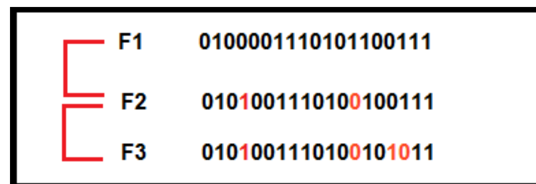


Figure 8. Three different bit vectors showing approximate macro co-change

2.1.3. *File Stability*

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

In ArgoUML: Macocha identifies 478 idle files and 1,143 changed files.

2.1.4. *Change Patterns*

Similar profiles grouped together represent occurrences of the Asynchrony and Dephase change patterns. Idles files do not change in any change period after their introduction into the program. Thus, we do not consider this group of files because they are not useful for the co-change analysis due to their rare evolution.

A S_{MCC} is two or more changed files that change together, *i.e.*, that have identical profiles during the life of a program, as illustrated in Figure 6. 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.

Copyright © 2000 John Wiley & Sons, Ltd.

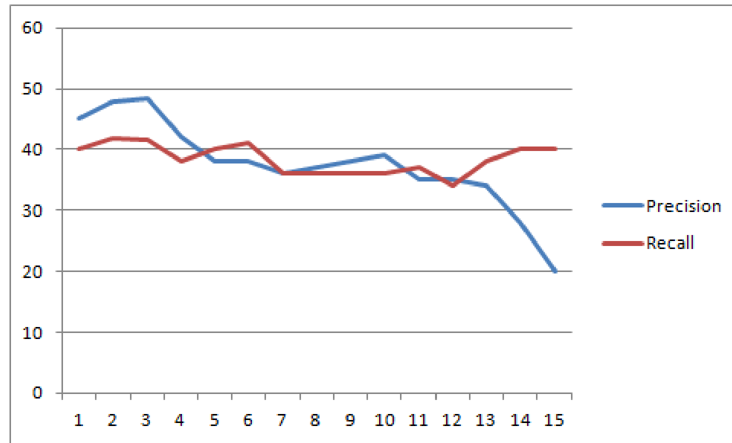


Figure 9. The mean of Precision and Recall achieved by Macocha with different values of D_H for the seven programs

Figure 7 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 ArgoUML: Macocha reports that `ProgressEvent.java` and `TestActionAddEnumerationLiteral.java` followed the Dephase change pattern with $s = 2$; and, `ProgressEvent.java` and `IConfigurationFactory.java` are in a DMCC with $s = 3$. Previous approaches cannot detected 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 differences 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 profile a and b is equal to the number of ones in the the vector $a \oplus b$.

After analysing several values of D_H between two profiles in different programs, 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 profiles are similar if their Hamming distance is less then three ($D_H < 3$).

Figure 8 illustrate that `F1` and `F2` are in approximate macro co-change with $D_H < 3$; `F2` and `F3` are in approximate MCC with $D_H < 3$; and, `F1` and `F3` are in a approximate MCC 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 MCC with $D_H < 2$;

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using smrauth.cls

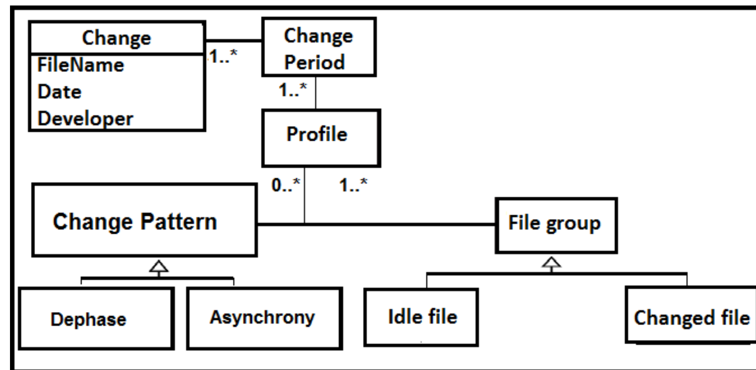


Figure 10. Meta-model of our data

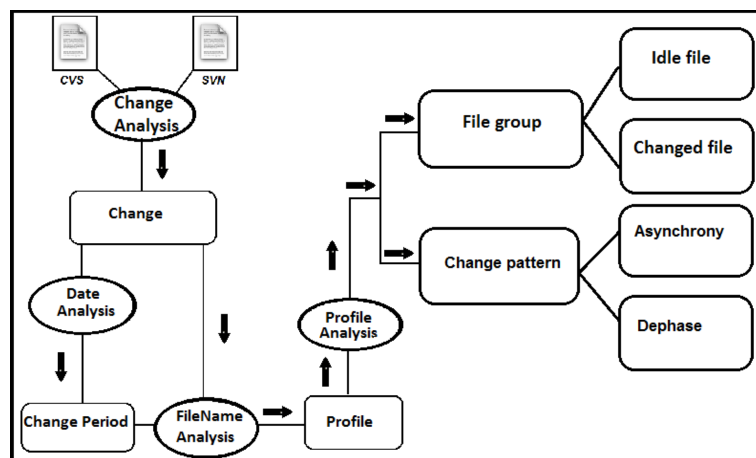


Figure 11. Analysis-process

2.2. Data Model, Implementation, and Outputs

Figure 10 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 11 illustrates the concrete process of Macocha. It takes as input a CVS/SVN change log. First, Macocha calculates the duration of different change periods using the k-nearest neighbor algorithm. Second, it groups changes in adequate change periods. Third, it creates a profile that describes the evolution of each file in each change period. fourth, it uses these profiles to compute the stability of the files and,

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using smrauth.cls

then, to identify changed files. Finally, Macocha detects macro co-changing files (changed files that follow the Asynchrony change pattern) and dephase macro co-changing files (changed files that follow the Dephase change pattern).

Macocha returns the following sets of occurrences of change patterns:

- S_{MCC}, the set of macro co-changing files with identical profiles in a program;
- S_{DMCC}, the set of dephase macro co-changing files identified when shifting profiles by s change periods with $s \in [0, 5]$;
- S_{MCCH}, 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_{DMCCH}, the set of approximate dephase macro co-changing files identified when shifting profiles by s change periods and by using the Hamming distance with $D_H \in]0, 3[$.

Macocha weighs, also, changes according to their distance in time: files co-changing frequently in the past but not in recent times may be less interesting than files having recently changed together. Thus, Macocha converts the bit vectors of files following a same change pattern to a decimal number, compare these decimals numbers to report the set of occurrences of change patterns including the most recently changed files (because as files are changed more recently, the conversion will naturally lead to greater decimal numbers). Because they are out of the scope of this paper, future work includes empirical studies of the usefulness for developers of ranking occurrences of the Asynchrony and Dephase change patterns.

3. Empirical study

Following GQM [BW84], the goal of our study is to show that our Macocha can identify occurrences of the Asynchrony and Dephase change patterns and that these occurrences describe interesting evolution phenomena. Our purpose is to bring generalisable, quantitative evidence on the existence of the Asynchrony and Dephase change patterns. The quality focus is the reduction of maintenance costs by detecting and using 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 five research questions:

- RQ1: How does Macocha compare to previous work (UMLDiff) in terms of changed files?
- RQ2: How does Macocha compare to previous work (association rules) in terms of precision and recall?
- RQ3: What is the precision and recall of Macocha when detecting occurrances of the Asynchrony pattern?
- RQ4: Are there occurences of Dephase change patterns in programs?
- RQ5: What is the usefulness of Dephase change patterns?

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using smrauth.cls

	ArgoUML	FreeBSD	JFreeChart	Openser	SIP	XalanC	XercesC
Languages	Java	C	Java	C	Java	C++	C++
Numbers of Versions	9	11	5	5	16	13	14
Numbers of Files	1,621	500	1,106	383	1,693	390	396

Table I. Descriptive statistics of the object programs

3.2. Objects

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++. We use these programs because they are open source, have been used in previous work [CCCDP] [ZBLL06], 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[5] 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 2007-02-19 to 2009-02-19. In this period, ArgoUML has gone through over 9 major versions.

FreeBSD[6] 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.

JFreeChart[7] is a Java open-source framework to create complex charts in a simple way. We analyse the evolution of this program from2008-02-13 to 2010-02-09. In this period, JFreeChart has gone through five versions.

Openser[8] is an open source implementation of a SIP server, licensed under the GNU General Public License. This program can be used as : SIP registrar server, SIP router, SIP redirect server, etc. It can be used, also, in small systems, for example in embedded systems 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 2007-06-04. In this period, Openser has gone through five versions.

SIP Communicator[9] 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.

[5] http://argouml.tigris.org/
[6] http://www.freebsd.org/
[7] http://www.jfree.org/
[8] http://www.opensips.org/
[9] http://www.sip-communicator.org/

Copyright © 2000 John Wiley & Sons, Ltd.

XalanC[10] 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 1999-02-21 to 2001-12-20. In this period, XalanC has gone through over 13 major versions.

XercesC[11] is a collection of software libraries for parsing, validating, serialising, and manipulating XML. We analyse the evolution of this program from its publishing in 99-11-09 for a period of two years. In this period, XercesC has gone through 14 versions.

3.3. Analyses

To answer our research questions, we apply Macocha to different object 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 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. Second, we compare the results of Macocha with those of the state-of-the-art approach for detecting co-changing files [ZWDZ04], which uses association rules. 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 by the association rules-based approach [ZWDZ04] is indeed a dependency link, using external information from bug reports, requirement descriptions, and mailing lists. We validate, also, each occurrence of $DMCC$ and MCC patterns found by Macocha in the analysed programs by studying their static relationships (such as use relations, inheritance relations, and so on). For program 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. Thus, for programs considered in this study and written with other language (FreeBSD, Openser, XalanC, and XercesC), we investigate static relationships among files following the Asynchrony and Dephase change patterns by manual source-code verification.

We thus report a quantitative analysis in accordance with the state of the art and a qualitative analysis in accordance with external information and a static analysis. We also report and discuss the number of occurrences of the (approximate) Asynchrony and Dephase change patterns.

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.

[10] `http://xml.apache.org/xalan-c/`
[11] `http://xerces.apache.org/xerces-c/`

Copyright © 2000 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

	Training set				Testing set			
	Start Date	End Date	# Transactions	# CPs	Start Date	End Date	# Transactions	# CPs
ArgoUML	07-02-19	09-02-19	4718	290	09-02-22	11-02-21	2225	191
FreeBSD	07-11-08	09-09-22	23944	85	09-12-21	11-10-31	26201	73
JFreeChart	08-02-13	10-02-09	1555	131	10-02-16	12-02-13	197	24
Openser	05-06-14	07-06-04	2321	247	07-06-05	09-06-15	3639	281
SIP	06-12-11	08-12-08	2870	261	08-12-09	10-12-09	3230	307
XalanC	99-12-21	01-12-20	2242	219	01-12-20	03-12-28	1379	165
XercesC	99-11-09	01-11-09	1820	204	01-11-12	03-11-08	2151	227

Table II. Internal evaluation of Macocha
(CPs: Change periods)

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

Table III. Cardinalities of the sets obtained in
the empirical study

4. Study Results and Discussions

We now present the results of our empirical study. Table III and Figure 13 summarises the cardinalities of the sets obtained by applying Macocha.

4.1. RQ1: How does Macocha compare to previous work (UMLDiff) in terms of changed files?

Macocha groups different commits in programs into change periods detected by the k-nearest neighbor algorithm. Figure 12 show the different change periods detected in each program. 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, detected in ArgoUML. We also observe that, in each program, change periods detected by the KNN algorithm have not the same duration. For example, the duration of the change periods detected in FreeBSD varies between 1 millisecond and 38 hours 48 minutes 42 seconds 747 ms.

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 programs. This analysis involves also eliminating idles files because they do not change in any change period after their introduction into the program. Therefore, they cannot participate in change patterns.

Because we want to distinguish idle from changed files, we group together the files identified as short-lived and active by UMLDiff and compare the sets provided by UMLDiff and by Macocha and

Copyright © 2000 John Wiley & Sons, Ltd.

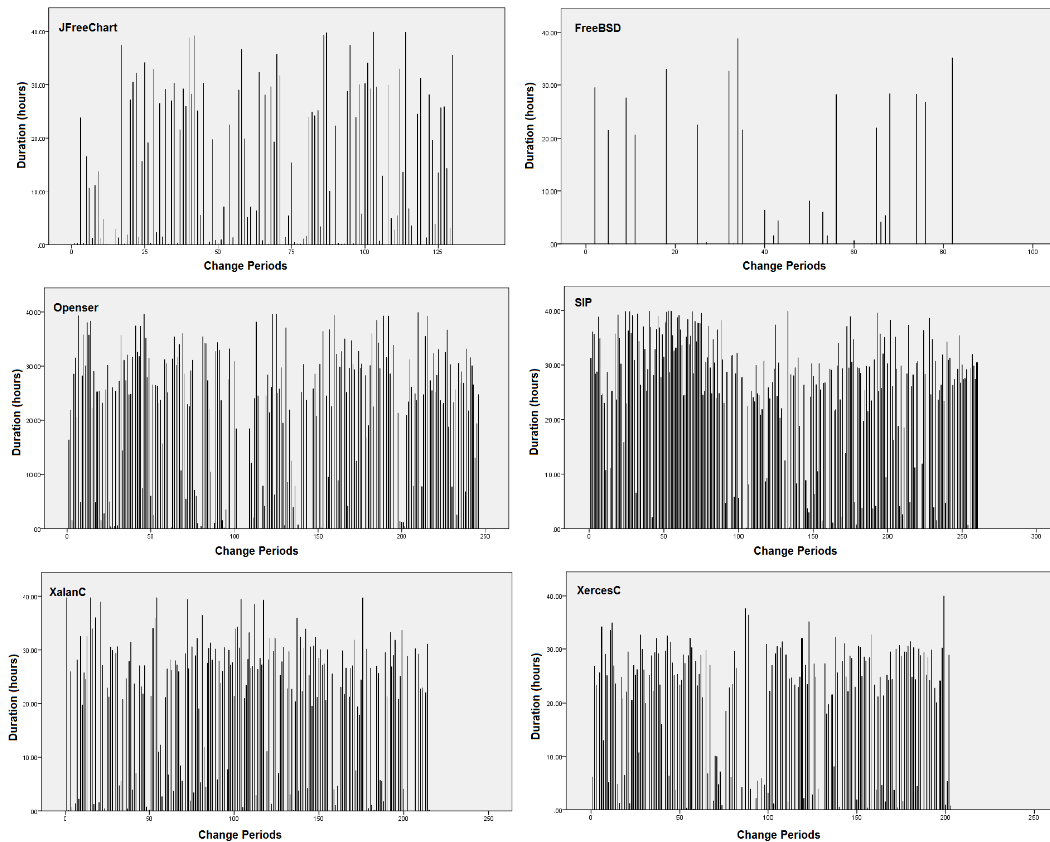


Figure 12. Change periods durations in different programs detected by the KNN algorithm

find that they are identical. For example, as shown in Table IV, Macocha finds 1,143 changed files in ArgoUML, identical to the UMLDiff $414+ = 729$ short-lived and active files.

Table IV reports the number of idle, short-lived, and active files found by UMLDiff in the object-oriented object 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 program developed with non object-oriented programming languages (FreeBSD and Openser developed in C) because it cannot create their UML-like representations. In the contrary to UMLDiff, Macocha can analyse file stability for any program, providing that their CVS/SVN repositories are available.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using smrauth.cls

		Idle Groups	Changed Groups
ArgoUML	Idle Clusters	478	0
	Short-lived Clusters	0	414
	Active Clusters	0	729
JFreeChart	Idle Clusters	398	0
	Short-lived Clusters	0	43
	Active Clusters	0	665
SIP	Idle Clusters	314	0
	Short-lived Clusters	0	742
	Active Clusters	0	637
XalanC	Idle Clusters	66	0
	Short-lived Clusters	0	122
	Active Clusters	0	202
XercesC	Idle Clusters	40	0
	Short-lived Clusters	0	170
	Active Clusters	0	186

Table IV. Cardinality of Macocha sets (idle groups and changed groups) in comparison to UMLDiff [XS05a]

Finally, Macocha computes file stability in few minutes (unlike UMLDiff, which takes few hours [XS05b]) because it does not create UML-like representations of the programs before performing its analysis. In the following, we present some examples from the object programs:

In JFreeChart: In two years of maintenance, Macocha analyse 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 confirm that this file belong to an idle cluster.

We detect, also, 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 confirm that this file belong to an active cluster.

In FreeBSD: We find 302 idle files. For example, `kvmproc.c` was modified in one change period in two years.

We detect also 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 can not analyse file stability in programs written in C.

> We thus answer RQ1: How does Macocha compare to previous work (UMLDiff) in terms of changed files? as follows: Macocha detects changed files with the same precision as UMLDiff but with a better recall to find occurrences of the Asynchrony and Dephase change patterns in a program, whatever its programming language.

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 MCCs sets) and report them. We compare the S_{MCCH} found by Macocha with the co-changing files found by an approach based on association rules [ZWDZ04] (see also [CCCDP]), which uses the Apriori algorithm

Copyright © 2000 John Wiley & Sons, Ltd.

	Macocha		Association Rules	
	Precision	Recall	Precision	Recall
ArgoUML	49%	30%	28%	29%
FreeBSD	19%	40%	11%	40%
JFreeChart	16%	33%	15%	33%
Openser	51%	32%	50%	32%
SIP	50%	55%	50%	52%
XalanC	82%	34%	79%	33%
XercesC	72%	56%	58%	46%

Table V. Internal evaluation of Macocha in comparison to an approach based on association rules [ZWDZ04]

[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] and a minimum support of 2 to compare association rules and our approach. We denoted the set of co-changing files found by an approach based on association rules [ZWDZ04] as S_{AR}.

We thus 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 II. 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}.

For the seven programs, we observe that Macocha improves precision and recall over the approach based on association rules, as shown in Table V. 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%. We observe that the Apriori algorithm generates high support sets of rules that are later checked for high confidence. Thus, high confidence rules with low support are not generated [AS94], which could lead to missed co-changing files.

We thus 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.

4.3. RQ3: What is the precision and recall of Macocha when detecting occurrances of the Asynchrony pattern?

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 [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 positive; (3) If the training set contained false positives or negatives, they cannot be detected using the testing set.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using smrauth.cls

	Macocha vs. Association Rules	
	Precision	Recall
ArgoUML	94%	98%
FreeBSD	82%	100%
JFreeChart	65%	96%
Openser	95%	89%
SIP	98%	100%
XalanC	100%	99%
XercesC	100%	100%

Table VI. External evaluation of Macocha when using the results of an approach based on association rules [ZWDZ04] as oracle

To overcome the limits of an internal validation and to validate change patterns 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.

We also apply a static analysis to validate the MCCs 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]. Thus, for each (approximate) occurrences of the Asynchrony change pattern, we validate the dependencies among their files by detecting their static relationships. If we cannot detect such static relationships, we check for other external information from bug reports, mailing lists, and so on to validate the MCCs detected by Macocha.

Thus, for each set returned by the association rules-base approach, if an identical set is returned by Macocha, 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 JFreeChart, all the sets detected using association rules are detected by Macocha except nine sets. We detect static relationships among files of seven sets from the nine missed sets and we validate the dependencies among the files of last two sets using bug reports in the Bugzilla of JFreeChart.

Table VII reports, under the External Information header, the 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. 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]. It 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.

In the following, we describe some (approximate) occurrences of the Asynchrony change pattern that are missed by the previous approach and justify why there are missed and why it is important to detect them. We choose theses 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 detect a static

Copyright © 2000 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

	External Validation of S_{MCCH} - S_{AR}	
	Precision	Recall
ArgoUML	100%	99%
FreeBSD	88 %	100%
JFreeChart	93 %	100%
Openser	100%	100%
SIP	100%	100%
XalanC	100%	100%
XercesC	100%	100%

Table VII. External evaluation 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

dependency among these two files. On the other hand, in the Bugzilla of ArgoUML, the bug ID 2552[12] 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 analyses for "Error Prevention" described in [ZWDZ04], we cannot find co-change dependencies among them. Thus, we could not explain and-or predict bugs in relation with theses two files.

JFreeChart: `AbstractXYDataset.java` and `RenderAttributes.java` were in MCC. As confirmed in the Bugzilla of JFreeChart by the bug ID 1654215[13] 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 `mungady` in a time-window of more than few minutes. Thus, by applying the association rules-based approach described in [ZWDZ04], we cannot not find that these files are co-changing. Thus, we could not give to the developer the necessary knowledge about dependencies among these two files to maintain them properly.

We thus answer RQ3: What is the precision and recall of Macocha when detecting occurrances of the Asynchrony pattern? as follows: the occurrences of the (approximate) Asynchrony change pattern found by Macocha are valid and brings interesting information to developers to prevent bugs.

4.4. RQ4: Are there occurences of Dephase change patterns in programs?

No previous approach can detect files maintained with similar trends and some given shifts in time. We validate the existence of occurrences of the (approximate) Dephase change patterns by detecting static relationships among their files. We also validate these occurrences using external information.

[12] `http://argouml.tigris.org/issues/show_bug.cgi?id=2552`
[13] `http://sourceforge.net/tracker/index.php?func=detail&aid=1654215&group_id=15494&atid=115494`

Copyright © 2000 John Wiley & Sons, Ltd.

Figure 13 illustrates the number of occurrences of the (approximate) Dephase change pattern detected and validated using external information and static analysis, in each program.

Macocha can detect occurrences of the (approximate) Dephase change pattern with several values of shift s. After analysing different sets of DMCCs detected in the seven object programs, we observed that the number of occurrences of the (approximate) Dephase change pattern detected by Macocha and validated by external information decreases from $s = 3$ and is close to 0 from $s = 5$, as shown in Figure 13. Thus, in our study, we detect DMCCs for $s \in [0, 5]$ to obtain accurate sets of results.

We now report some typical occurrences of the (approximate) Dephase change pattern from different programs with different value of shift s. We choose theses 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 are related in a lengthy the message from `Ian Smith` about “IPFW transparent VS dummynet rules”. Thus, we confirm dependencies among these two files by external information.

In SIP: We find that `Html2Text.java` and `FileTransfReceiveListener.java` were changed systematically with five shift change period in two years. Thus, they followed the Dephase change pattern with shift $s = 5$. These two files implement the same feature[14]: “Instant Messaging”. By performing a static analysis, we detect a static dependency among these two files. Thus, we validate the occurrence of the Dephase change pattern formed by these two files.

In XercesC: We find 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[15] from `Wesley Smal` on April 1, 2009 about “a legitimate bug with the time of day” states that these two files are related.

We thus answer RQ4: Are there occurences of Dephase change patterns in programs? as follows: in all the object program, Macocha detects valid occurrences of the (approximate) Dephase change pattern.

4.5. RQ5: What is the usefulness of Dephase change patterns?

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

4.5.1. Management of Development Teams

If two files follow the same occurrence of a dephase change pattern, they should ideally be maintained by the same team of developers to minimise the risks of introducing bugs in the future. The team of

[14] `http://www.jitsi.org/index.php/Main/Features`
[15] `http://markmail.org/message/a5secbiwkgxtegxb`

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using smrauth.cls

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 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[16] [17] [18] 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.

4.5.2. 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 [ZWDZ04], 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.

4.5.3. 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 [ZWDZ04], 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.

We thus answer RQ5: What is the usefulness of Dephase change patterns? as follows: in all the object programs, the occurrences of the (approximate) Dephase change pattern detected by Macocha could be useful to ease maintenance and reduce its costs.

[16] `http://argouml.tigris.org/issues/show_bug.cgi?id=1957`
[17] `http://argouml.tigris.org/issues/show_bug.cgi?id=2926`
[18] `http://argouml.tigris.org/issues/show_bug.cgi?id=4604`

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using smrauth.cls

5. Discussions

With our approach, we detect several occurrences of the (approximate) Asynchrony and Dephase change patterns in seven 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 proportions in each program. We observe that the numbers of MCCs and DMCCs found in the programs developed in Java (ArgoUML, JFreeChart, and SIP) are greater than the number of MCCs and DMCCs found in programs developed in C or C++ (see Table III and Figure 13). 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, Openser, XalanC, XercesC are the smallest object programs (having less than 400 files). Thus, we also apply our approach to detect (dephase) macro co-changes on fewer C and C++ files than Java files, which could thus explain 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.

5.1. Threats to the Validity

Some threats limit the validity of the result 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, MCCs and DMCCs exist and are corroborated by external sources of information (bug reports and others). 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. The change logs 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/`.

External Validity: We performed our study on seven 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.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using smrauth.cls

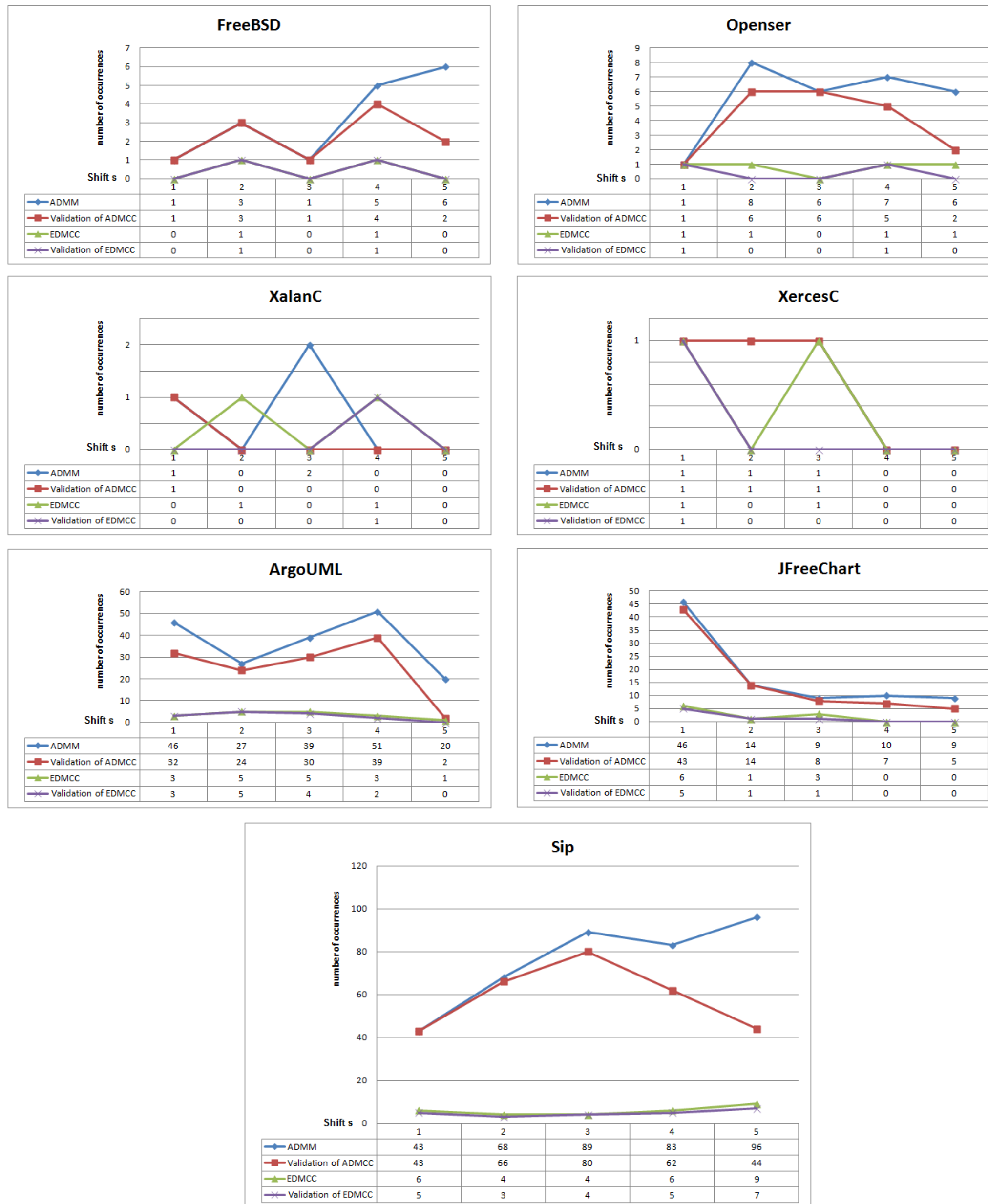


Figure 13. Evolution of the number of occurrences of dephase change patterns detected and validated by static analyses for different values of shift s

Copyright © 2000 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

6. Related Work

The concepts of MCCs and DMCCs relate our work to that on file stability, co-change and change patterns, and change propagation. We present related work and discuss these works 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, Kpodjedo *et al.* in [KRGA09] and [KRG$^+$] 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 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, thus, that likely constitute the stable backbone of programs. As other 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).

Discussion: The Error Tolerant Graph Matching algorithm and UMLDiff take few hours to analyse file stability for the seven 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.

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

Copyright © 2000 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

without majors modifications in it implementation, it will be identified by our approach as "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 algorithm uses co-change history in CVS and avoids the source code dependency parsing process.

Discussion: Approaches based on association rules [YMNCC04] and [ZWDZ04], compute only the frequency of co-changed files in the past and omits many other cases, *e.g.*, files that co-change with some shifts among change period. In Section 4, 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.

Ceccarelli *et al.* [CCCDP10] and Canfora *et al.* [CCCDP] 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.

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).

Discussion: In [CCCDP10] and [ARV05], if 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. Indeed, their approaches could find files having very similar maintenance evolution history but they did not present a tool to detect MCCs and DMCCs. In this paper, we present a tool to detect such change patterns and we showed their usefulness of.

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: In this paper, we introduce two novel change patterns inspired from co-changes and using the concept of change periods. We define the asynchrony change pattern that describes artefacts that co-change in large time intervals, and the dephase change pattern that always happen with the same shifts in time.

Copyright © 2000 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

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 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 [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.

Finally, 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.

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 and may involve several files that are in MCCs or in DMCCs but they do not allow to differentiate between these two change patterns. All these approaches grouped change couplings created by the same author and have the same log message; thus, they can not detect asynchrony and dephase change patterns.

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 storyboard, 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.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using smrauth.cls

Discussion: However, Xing and Stroulia [XS07], 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.

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, in [SBA06], the authors defined the Asynchrony change pattern as common and recurring modifications of programs' files in time.

In this paper, we reused the Asynchrony change pattern and introduced 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 program's files appropriately. We proposed an approach, Macocha, which mines software repositories and uses several algorithms and techniques, such as the k-nearest neighbor 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 on seven systems: ArgoUML, FreeBSD, JFreeChart, Openser, SIP, XalanC, and XercesC, and showed that Macocha has a better precision and recall than the state-of-the-art approached based on association rules [YMNCC04, ZWDZ04]. Qualitatively, we used external information and static analysis to show that detected MCCs and DMCCs explain real, important evolution phenomena. We also thus showed that occurrences of the Asynchrony and Dephase change patterns do exist and can help in explaining bugs, managing development teams, and performing traceability analysis.

Thus, this paper extended our previous work [JGHA11] with the following contributions. First, we described in more details our approach, Macocha. Second, we used the k-nearest neighbor algorithm (KNN) to group changes into change periods and therefore to determine automatically the duration of the different change periods in each program. Third, we performed an extensive validation of Macocha on seven programs (three more programs: JFreeChart, Openser and XercesC) developed with three different languages: C, C++, and Java. Fourth, we studied the variations in precision and recall of our approach when using different values of its parameters. Finally, we provided evidence on the relevance of the Asynchrony and Dephase change patterns.

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) evaluate the consistency and the usefulness of change patterns' occurrences including files recently changed over other occurrences (4) relating Asynchrony and Dephase change patterns with program quality and external software characteristics, such as change proneness. Also, future work 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.

Copyright © 2000 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

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

[ARV05] . Giuliano Antoniol, Vincenzo Fabio Rollo, and Gabriele 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*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[AS94] . Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[BH06] . Dirk Beyer and Ahmed 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.

[BKZ10] . Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. Codebook: discovering and exploiting relationships in software repositories. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 125–134. ACM, 2010.

[BW84] . Victor R. Basili and David M. Weiss. A methodology for collecting valid software engineering data. *Software*, 10(6):728–738, 1984.

[CC05] . Gerardo Canfora and Luigi Cerulo. Impact analysis by mining software and change request repositories. In *Proceedings of the 11th IEEE International Software Metrics Symposium*, page 29, Washington, DC, USA, 2005. IEEE Computer Society Press.

[CCCDP] . Gerardo Canfora, Michele Ceccarelli, Luigi Cerulo, and Massimiliano 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*, pages 1–10, Washington, DC, USA. IEEE Computer Society Press.

[CCCDP10] . Michele Ceccarelli, Luigi Cerulo, Gerardo Canfora, and Massimiliano Di Penta. An eclectic approach for change impact analysis. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 163–166, New York, NY, USA, 2010. ACM Press.

[Das91] . B.V. Dasarathy. Nearest neighbor ({NN}) norms:{NN} pattern classification techniques. 1991.

[DLL09] . Marco D'Ambros, Michele Lanza, and Mircea Lungu. Visualizing co-change information with the evolution radar. *Transactions on Software Engineering*, 35(5):720–735, 2009.

[FPG03] . Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, pages 23–, Washington, DC, USA, 2003. IEEE Computer Society.

[GA08] . Yann-Gaël Guéhéneuc and Giuliano Antoniol. DeMIMA: A multi-layered framework for design pattern identification. *Transactions on Software Engineering (TSE)*, 34(5):667–684, September 2008. 18 pages.

[GAA04] . Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In Doug C. Schmidt, editor, *Proceedings of the 19th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 301–314. ACM Press, October 2004. 14 pages.

[Ger06] . Daniel M. German. An empirical study of fine-grained software modifications. *Empirical Softw. Engg.*, 11, September 2006.

[GHJ98] . Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance*, pages 190–, Washington, DC, USA, 1998. IEEE Computer Society.

[HAM50] . R. W. HAMMING. Error detecting and error correcting codes. *BELL SYSTEM TECHNICAL JOURNAL*, pages 147–160, 1950.

[Hat07] . Les Hatton. How accurately do engineers predict software maintenance tasks? *Computer*, 40:64–69, February 2007.

[HH04] . Ahmed E. Hassan and Richard C. Holt. Predicting change propagation in software systems. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 284–293, Washington, DC, USA, 2004. IEEE Computer Society.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using smrauth.cls

[JGHA11] . F. Jaafar, Y.G. Guéhéneuc, S. Hamel, and G. Antoniol. An exploratory study of macro co-changes. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 325–334. IEEE, 2011.

[KR11] . David Kawrykow and Martin P. Robillard. Non-essential changes in version histories. In *Proceeding of the 33rd International Conference on Software Engineering*, ICSE '11, pages 351–360, New York, NY, USA, 2011. ACM.

[KRG+] . S. Kpodjedo, F. Ricca, P. Galinier, G. Antoniol, and Y.G. Guéhéneuc. Studying software evolution of large object-oriented software systems using an etgm algorithm. *Journal of Software Maintenance and Evolution: Research and Practice*.

[KRGA09] . Segla Kpodjedo, Filippo Ricca, Philippe Galinier, and Giuliano Antoniol. Recovering the evolution stable part using an ecgm algorithm: Is there a tunnel in mozilla? In *CSMR*, pages 179–188, 2009.

[KZP07] . SB Kotsiantis, ID Zaharakis, and PE Pintelas. Supervised machine learning: A review of classification techniques. *FRONTIERS IN ARTIFICIAL INTELLIGENCE AND APPLICATIONS*, 160:3, 2007.

[LB85] . M. M. Lehman and L. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[LD02] . Michele Lanza and Stéphane Ducasse. Understanding software evolution using a combination of software visualization and software metrics. 2002.

[LR03] . James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 308–318, Washington, DC, USA, 2003. IEEE Computer Society.

[MFH02] . Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11:309–346, July 2002.

[RWBW90] . Brian Wilkerson Rebecca Wirfs-Brock and Lauren Wiener, editors. *Designing Object-Oriented Software*. Prentice Hall, 1990.

[SBA06] . Yann-Gaël Guéhéneuc Salah Bouktif and Giuliano Antoniol. Extracting change-patterns from cvs repositories. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pages 221–230, Washington, DC, USA, 2006. IEEE Computer Society.

[VKvRvV09]. Adam Vanya, Steven Klusener, Nico van Rooijen, and Hans van Vliet. Characterizing evolutionary clusters. In *Proceedings of the 16th Working Conference on Reverse Engineering*, Washington, DC, USA, 2009. IEEE Computer Society.

[XS05a] . Zhenchang Xing and Eleni Stroulia. Analyzing the evolutionary history of the logical design of object-oriented software. *Transactions on Software Engineering*, 31:850–868, 2005.

[XS05b] . Zhenchang Xing and Eleni Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th International Conference on Automated Software Engineering*, New York, NY, USA, 2005. ACM Press.

[XS07] . Zhenchang Xing and Eleni Stroulia. Bottom-up design evolution concern discovery and analysis. Technical report, 2007.

[YMNCC04]. Annie T. T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting source code changes by mining change history. *Transactions on Software Engineering*, 30(9):574–586, 2004.

[ZBLL06] . Thomas Zimmermann, Silvia Breu, Christian Lindig, and Benjamin Livshits. Mining additions of method calls in argouml. In *Proceedings of the International Workshop on Mining Software Repositories*. ACM Press, 2006.

[ZWDZ04] . Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.

[ZWG+08] . Yu Zhou, Michael Würsch, Emanuel Giger, Harald C. Gall, and Jian Lü. A bayesian network based approach for change coupling prediction. In *Proceedings of the 15th Working Conference on Reverse Engineering*, pages 27–36, Washington, DC, USA, 2008. IEEE Computer Society.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using smrauth.cls