Co-evolution Analysis of Production and Test Code by Learning Association Rules of Changes

László Vidács
MTA-SZTE Research Group on Artificial Intelligence
University of Szeged
Hungary
lac@inf.u-szeged.hu

Martin Pinzger
Software Engineering Research Group
University of Klagenfurt
Austria
martin.pinzger@aau.at

***Abstract*—Many modern software systems come with automated tests. While these tests help to maintain code quality by providing early feedback after modifications, they also need to be maintained. In this paper, we replicate a recent pattern mining experiment to find patterns on how production and test code co-evolve over time. Understanding co-evolution patterns may directly affect the quality of tests and thus the quality of the whole system. The analysis takes into account fine grained changes in both types of code. Since the full list of fine grained changes cannot be perceived, association rules are learned from the history to extract co-change patterns. We analyzed the occurrence of 6 patterns throughout almost 2500 versions of a Java system and found that patterns are present, but supported by weaker links than in previously reported. Hence we experimented with weighting methods and investigated the composition of commits.**

***Index Terms*—software evolution, change analysis, machine learning, co-evolution patterns, testing**

I. Introduction

Software development produces a tremendous amount of data about the development process and the software itself that are recorded in software repositories, such as GitHub and Bitbucket. They track, for instance, changes to files, in particular which developer changed which file when and how, and bug reports submitted by users. To extract the valuable part of the data from the evolution point of view, repository mining techniques are usually applied. In this research project, we employed repository mining techniques combined with machine learning to assess the co-evolution of production and test code. Balanced co-evolution of source code and other development artifacts is an important factor in maintaining software quality of evolvable systems [1], [2]. In recent years, as large amount of test code also needs significant effort to maintain, test suite evolution received increased interest [3]. Today test code is treated as a first class citizen in software projects, which urges the in depth understanding of the relation of production and test code. This fosters traceability research [4], [5], [6] and makes the co-evolution analysis of production and test code an emerging topic [7], [8].

The aim of this paper is to analyze the history of production and test code changes to reveal whether co-evolution patterns in project history are followed in order to keep software quality at the desired level. The research approach followed in this work is based on a recent study by Marsavina *et al.* [8]. We applied repository mining, test coverage analysis and association rule learning techniques to obtain patterns of co-evolution. The main point of this study is the replication of the previous experiment, with additional analysis to give deeper insight into how co-evolution happens. We provide the following contributions in this paper:

- Replication of Marsavina *et al.* [8] on one system.
- Extended analysis of project properties, experiments with change weighting methods and analysis of composition of commits.

II. Co-evolution Analysis

A. Overview

This work is based on the study by Marsavina *et al.* [8] on mining co-evolution patterns. In that study the authors identified production and test code co-evolution patterns through aggregating fine-grained changes and applying association rule learning on changes measured through the project history. The direction of the analysis is from production classes to test classes. We replicated the analysis method, in which the main question is: *When developers change production code, do they change test classes as well (and how)?*

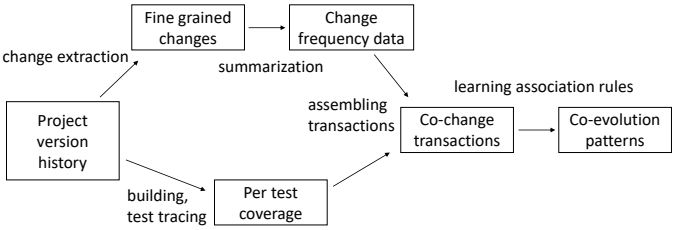

Fig. 1. Analysis process overview

The overview of the analysis process is shown in Figure 1. First, fine-grained change information is extracted form the version control system for each commit. Second, dynamic analysis is used to obtain test coverage data for each commit. Based on these two sources, we use association rule learning to find regular patterns of developer behaviour, for example whether introducing a new class in a commit implies that a new test class in also introduced. The link between production

TABLE I
OVERVIEW OF THE ANALYZED PROJECT

Project	# Versions analyzed	First version # Classes	# Prod. Methods	# Test Methods	Date	Last version # Classes	# Prod. Methods	# Test Methods	Date	# Versions Non-building due to Test fail.	# Versions Non-building due to Cov. fail.
commons-lang	2470	115	2020	1622	2009-10-13	206	3025	3880	2016-05-29	80	16

TABLE II
OVERVIEW OF THE ANALYZED PROJECT FROM MARSAVINA *et al.*

Project	# Versions analyzed	First version # Classes	# Prod. Methods	# Test Methods	Date	Last version # Classes	# Prod. Methods	# Test Methods	Date	# Versions Non-building due to Test fail.	# Versions Non-building due to Cov. fail.
commons-lang (Mars.)	3856	31	373	318	2002-12	177	2442	2851	2014-02	54	-

and test classes is provided by the dynamic analysis, since during the rule mining we consider only those test classes which in fact exercise (cover) the changed production class.

Besides our base study [8], there were similar approaches applied for co-evolution analysis. Our study goes back to earlier works by Zaidman *et al.* [7], [9]. Pinto *et al.* provided a large-scale study of test suite evolution emphasizing the need for understanding how tests evolve to aid other purposes like automatic repair techniques [3], [10]. Marinescu *et al.* provided a framework that combines static and dynamic analysis of tests and their coverage throughout the version history for C/C++ programs [11]. We apply a similar approach for analyzing Java source code. Co-evolution analysis needs a proper link between tests and the related production classes. Rompaey *et al.* compared several test to code traceability methods, including the use of naming conventions and various structural and conceptual methods [6]. Ghafari *et al.* presented a method level traceability solution [12]. A remarkable solution is provided by Qusef *et al.*, which combines program slicing and information retrieval methods [4], [13]. Telea *et al.* [14] used code visualization for evolution analysis of production code. Ens *et al.* provided the ChronoTwigger tool to support visual analytics of test and code evolution [15].

In our study we rely on a state of the art solution to address the following research questions:

RQ1: Do production and test code evolve in sync?

RQ2: What kind of fine-grained co-evolution patterns between production and test code can be identified?

B. Project history

In this report we provide a detailed analysis of the `commons-lang` project. Table I shows the main characteristics of the analysis, while in Table II values from the original study can be seen. The tables report project sizes in the first and in the last analyzed versions for each study. The number of non-building versions is slightly higher in our case, but compared to all versions analyzed it is negligible. We also report the number of failed builds because of failing coverage measurements, which is a small number as well. In total 2470 versions were analyzed with more than 4 years overlap with the original study in the project history. We did not consider early versions since we used a more precise source code level instrumentation, for which maven integration was necessary.

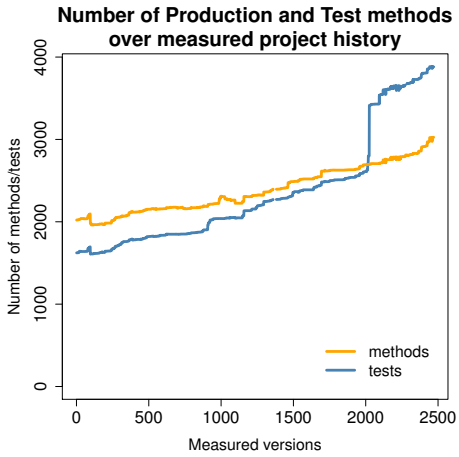

Fig. 2. History of the number of production and test classes

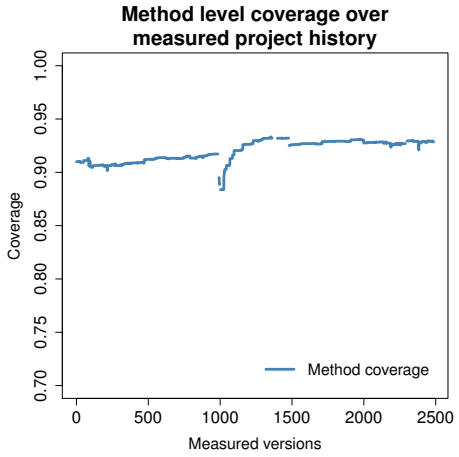

Fig. 3. History of the method level coverage

The analysis of the project history may be biased if there are too many exceptional events or large changes in the project workflow. In Figure 2 the history of the number of production

and test classes is shown. The addition of new production classes is balanced; and test classes are also added in parallel. The two curves follow each other in most of the time. There is one huge increase in the number of test classes, but this is a unique occasion. Another important aspect we measured is the global coverage of all tests. In Figure 3 the method level coverage ratio is depicted. Note that the coverage ratio was already relatively high at the beginning of our measurements, the scale of the *y* axis starts at 0.7 coverage. There is a slight coverage increase over the time, which is the sign of increasing effort invested in testing, since the number of production classes also increased in that period of time. There is a small period with no successful coverage measurements, and with a huge decrease in method level coverage, but the developers of tests managed to balance it out in a short period of time.

The coverage measurements were done using the Clover tool. The test traces were processed using the SoDA toolchain [16], which handles the per test coverage matrix and is also able to handle test results and compute test suite metrics. Test coverage is used as the test-to-code traceability [6], [17] solution in our study.

C. Fine grained changes

We obtained fine grained changes using the ChangeDistiller [18] through a wrapper tool. According to the ChangeDistiller model, detailed changes belong to 10 change categories. When processing changes we distinguish production and test classes; both of them can contain any number of detailed changes. Change categories are listed in Table III divided into production and test class changes.

TABLE III
TOTAL NUMBER OF CHANGES IN PRODUCTION AND TEST CODE PER CHANGEDISTILLER CATEGORY

ChangeDistiller category	commons-lang	
	Prod	Test
ADDED_CLASS	198	257
REMOVED_CLASS	133	153
CLASS_DECLARATION	1153	1631
METHOD_DECLARATION	644	236
ATTRIBUTE_DECLARATION	504	321
BODY_STATEMENTS	6845	9425
BODY_CONDITIONS	934	68
COMMENTS	368	349
DOCUMENTATION	1247	225
OTHERS	7	0
Total	12033	12665
Test ratio	0.512795	

The distribution of changes both between categories and between the two types of classes are similar to the data found by Marsavina *et al.* Not surprisingly, changes in body statements happen most frequently. Important to note is that slightly more than half of the fine grained changes happened in test classes in this project. Although the measured interval in the project history is different, the finding of the base study holds in our analysis as well.

In Figure 4 the ratio of changed production and test classes is shown for the whole analyzed history. In the number of classes the high intensity of test changes can also be observed, which means that developers in this project take care of tests.

As already mentioned, the direction of the analysis is from production classes to test classes (test-driven projects need changes in this process). For each production class we checked if there are any test classes that were changed in the same commit. As Figure 5 shows in the second and third bars, more than half of the production classes are changed without co-changed test classes. These cases unfortunately hinder the analysis of co-evolution patterns. Although we found many small changes in the project history (as seen in Table III), there are many cases when the developers do not (immediately) initiate changes both in production and test code.

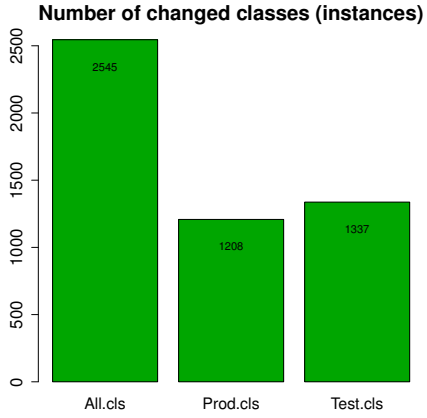

Fig. 4. Distribution of changed Java classes

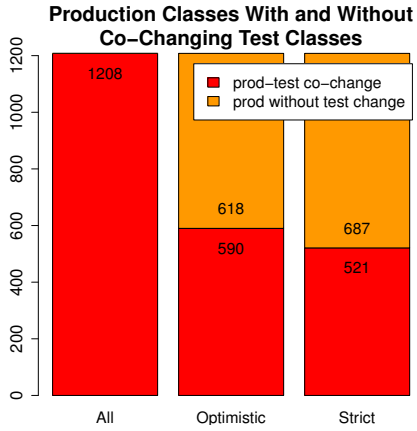

Fig. 5. Number of co-changes from the production classes perspective

III. MINING CO-EVOLUTION PATTERNS

The aim of the mining process is to obtain association rules between production and test classes. Association rule learning is a well-known data mining method for discovering regularities in large-scale transaction databases. This learning method has been successfully applied for software engineering problems as well [19], [9]. In our case, fine grained changes are considered as transaction items, while commit level project

TABLE IV
ASSOCIATION RULES MINED FROM EVOLUTION DATA

Rule	Production side		Test side	Commons-Lang			
Id	Rule LHS		Rule RHS	W1	W1'	W2	Marsavina *et al.*
1	ADDED_CLASS_P=YES	$\Longrightarrow$	ADDED_CLASS_T	**YES** 0.51 / 0.51	**SOMETHING** /	**YES** 0.51 / 0.51	**YES** 412 / 0.643
2	REMOVED_CLASS_P=YES	$\Longrightarrow$	REMOVED_CLASS_T	**YES** 0.56 / 0.56	**SOMETHING** /	**YES** 0.56 / 0.56	**YES** 569 / 0.998
3a1	CLASS_DECLARATION_P=LOW	$\Longrightarrow$	CLASS_DECLARATION_T	NONE 0.50 / 0.50	**SOMETHING** /	**SOMETHING** /	**NONE** 244 / 0.953
3a2	CLASS_DECLARATION_P=MED_LOW	$\Longrightarrow$	CLASS_DECLARATION_T	NONE 0.54 / 0.54	**SOMETHING** /	**SOMETHING** /	**LOW** 132/ 0.8
3a3	CLASS_DECLARATION_P=MED_HIGH	$\Longrightarrow$	CLASS_DECLARATION_T	NONE 0.55 / 0.55	**SOMETHING** /	**SOMETHING** /	**SOMETHING** /
3a4	CLASS_DECLARATION_P=HIGH	$\Longrightarrow$	CLASS_DECLARATION_T	**HIGH** 0.64 / 0.64	**HIGH** 0.64 / 0.64	**HIGH** 0.55 / 0.55	**HIGH** 85 / 0.658
4a1	METHOD_DECLARATION_P=LOW	$\Longrightarrow$	BODY_STATEMENTS_T	NONE 0.55 / 0.55	**SOMETHING** /	NONE 0.65 / 0.65	**SOMETHING** /
4a2	METHOD_DECLARATION_P=MED_LOW	$\Longrightarrow$	BODY_STATEMENTS_T	NONE 0.53 / 0.53	**SOMETHING** /	**SOMETHING** /	**SOMETHING** /
4a3	METHOD_DECLARATION_P=MED_HIGH	$\Longrightarrow$	BODY_STATEMENTS_T	NONE 0.67 / 0.67	NONE 0.67 / 0.67	NONE 0.75 / 0.75	**SOMETHING** /
4a4	METHOD_DECLARATION_P=HIGH	$\Longrightarrow$	BODY_STATEMENTS_T	NONE 0.68 / 0.68	NONE 0.68 / 0.68	NONE 0.56 / 0.56	MED_HIGH 37 / 0.616
5a1	ATTRIBUTE_DECLARATION_P=LOW	$\Longrightarrow$	BODY_STATEMENTS_T	NONE 0.63 / 0.63	NONE 0.63 / 0.63	NONE 0.90 / 0.90	**SOMETHING** /
5a2	ATTRIBUTE_DECLARATION_P=MED_LOW	$\Longrightarrow$	BODY_STATEMENTS_T	NONE 0.95 / 0.95	NONE 0.95 / 0.95	**SOMETHING** /	**SOMETHING** /
5a3	ATTRIBUTE_DECLARATION_P=MED_HIGH	$\Longrightarrow$	BODY_STATEMENTS_T	NONE 1.00 / 1.00	NONE 1.00 / 1.00	NONE 0.70 / 0.70	**SOMETHING** /
5a4	ATTRIBUTE_DECLARATION_P=HIGH	$\Longrightarrow$	BODY_STATEMENTS_T	NONE 0.98 / 0.98	NONE 0.98 / 0.98	NONE 0.63 / 0.63	**SOMETHING** /
6a1	BODY_CONDITIONS_P=LOW	$\Longrightarrow$	CLASS_DECLARATION_T	NONE 0.71 / 0.71	NONE 0.71 / 0.71	NONE 0.80 / 0.80	NONE 126/ 0.670
6a2	BODY_CONDITIONS_P=MED_LOW	$\Longrightarrow$	CLASS_DECLARATION_T	NONE 0.82 / 0.82	NONE 0.82 / 0.82	NONE 0.70 / 0.70	**SOMETHING** /
6a3	BODY_CONDITIONS_P=MED_HIGH	$\Longrightarrow$	CLASS_DECLARATION_T	NONE 0.86 / 0.86	NONE 0.86 / 0.86	NONE 0.76 / 0.76	**SOMETHING** /
6a4	BODY_CONDITIONS_P=HIGH	$\Longrightarrow$	CLASS_DECLARATION_T	NONE 0.84 / 0.84	NONE 0.84 / 0.84	NONE 0.73 / 0.73	**SOMETHING** /

history provides the transaction database. An example association rule is shown in Figure 6. This rule means that when a new production class is added to the code, then a new test class is also added in slightly more than half of the cases in project history. Since our question is whether changes in production code induce changes in test code, we work with rules where the first part is about production code changes and the implication is about test code changes.

```
ADDED_CLASS_PROD = YES ⟹ ADDED_CLASS_TEST = YES
(support=0.51)
```

Fig. 6. Association rule for added classes

The next step in the co-evolution mining process is to use the fine-grained change data to produce input for the association rule mining algorithm. For this purpose, changes are summarized by change category for each changed production class for each commit. Co-evolution rules are mined when there were test class changes as well within the same commit. To consider a changed test class it has to cover the given production class of the same commit, which is checked using the dynamic coverage matrix obtained using the Clover tool. Thus we produce changed production class to changed test class links, where change categories are computed for both types of classes. When there are more than one changed test classes that cover the production class, the test class changes are summarized by change type.

Association rule mining is used to mine patterns in such linked classes. This algorithm does not work on numeric results, so we need to discretize the number of changes for each change category. We compute the quartiles for the whole project history for each change type (both production and test classes are considered). For each class we use the `NONE`, `LOW`, `MED_LOW`, `MED_HIGH` and `HIGH` categories for `0`, `Q1`, `Q2`, `Q3` and `Q4` values respectively. In binary cases we use only `NO` and `YES` categories. After this step we produce the change vectors for each changed production class for each commit as shown in Figure 7. The vector consists of 7+7 elements, one element for each of the 7 main change categories for production (*_P*) and test (*_P*) classes. These vectors correspond to transactions in association rule mining terminology. Rules are mined using the apriori algorithm of the *arules* package of the *R* repository. To find patterns we set the support threshold to 0.5.

```
ADDED_CLASS_P = NO;              ADDED_CLASS_T = NO;
REMOVED_CLASS_P = NO;            REMOVED_CLASS_T = NO;
CLASS_DECLARATION_P = HIGH;      CLASS_DECLARATION_T = NONE;
METHOD_DECLARATION_P = NONE;     METHOD_DECLARATION_T = NONE;
ATTRIBUTE_DECL_P = NONE;         ATTRIBUTE_DECL_T = NONE;
BODY_STATEMENTS_P = HIGH;        BODY_STATEMENTS_T = NONE;
BODY_CONDITIONS_P = MID_LOW;     BODY_CONDITIONS_T = NONE;
```

Fig. 7. Co-change transaction example for association rule mining

The results of co-evolution analysis are summarized in Table IV. We reused the 6 types of rules from [8]. The left hand side of each association rule reflects changes in production code, while the right hand side shows whether linked test code changes exist in the project history. For each rule, the whole set of transactions is filtered for the given assumption of the production code. For example `Rule 3a1` checks whether `LOW` changes in production class declarations implies changes of test class declarations at 0.5 support level. The nature of the test change is given on the right hand side of the table. When none of the change categories reaches the desired support level, but still there is some link between production and test changes, `SOMETHING` is used (otherwise `NONE` would be the result).

TABLE V
CO-EVOLUTION PATTERNS

Pattern Id	Description	Presence
1/2	When a new production class is added/deleted, an associated test class is also created/deleted	✓/✓
3	When a new production method is created, one or more test cases addressing it are also developed	✓
4	When method-related changes occur in the production code, the tests are updated accordingly	Weak
5	When a field is added in the production code, the existing test cases are updated in order to address	Weak
6	Upon modifying conditional statements in methods from the production code, new test cases are created	?

We conducted several experiments adjusting the parameters of the transaction generation and rule mining. The `W1` column reports the results of measurements where the volume of the changes are not bound to quartiles, but represent local values within the commit. In case of `W1'` the confidence level is set to 0.6 as in the base study. The `W2` column shows the results of the quartile-based measurements, while the last column shows the base values from [8]. We also experimented with a rule mining method using only binary `YES/NO` values for all rules instead of the quartiles to double check the results.

Overall, we found weaker links than it was reported in the previous study. Table V shows the identified co-evolution patterns: 3 patterns confirmed, 2 patterns present but weakly supported and one pattern is missing. The `W1` column in Table IV gives insights into the co-change data, and our observation is that in these cases the `NONE` rules is too high to reproduce the base study. On the other hand, in the base study there were only `SOMETHING` results in these cases, indicating weak links as well. Thus, we consider the replication of the base study successful.

IV. DISCUSSION

To understand more details about the volume of the rules, we analyzed the change frequencies of fine-grained changes by change types provided by ChangeDistiller as seen in Figure 8. Except body statement changes, which is far the most frequent change type that occurs, even the median of change count is one. Higher numbers are mostly outliers. Not surprisingly, this causes that most of the quartiles are around 1 (see Table VI). Consequently, for our subject program, the binary approach (`NO` and `YES` categories) may be a good approximation.

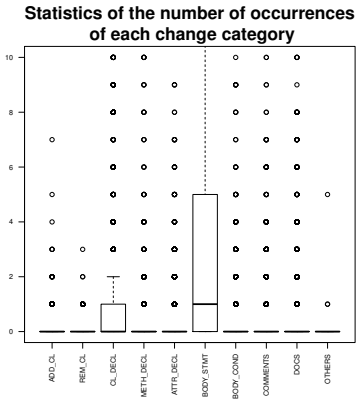

Fig. 8. Change frequency quartiles

TABLE VI
QUARTILES USED AS THRESHOLDS PER CHANGEDISTILLER CATEGORY

ChangeDistiller category	Q1	Q2	Q3
ADDED_CLASS	1	1	1
REMOVED_CLASS	1	1	1
CLASS_DECLARATION	1	2	3
METHOD_DECLARATION	1	2	3
ATTRIBUTE_DECLARATION	1	1	3
BODY_STATEMENTS	2	4	11
BODY_CONDITIONS	1	2	3
COMMENTS	1	2	3
DOCUMENTATION	1	1	3
OTHERS	1	1	1

Binary weighting analysis and the analysis of the change quartiles showed that in many cases there is no link between production and test class changes. We further analyzed the commit level co-change data since our analysis method can reason based on commits that contain both production and test classes.

Table VII and Figure 9 show details about the composition of commits from the co-change point of view. Test only commits (29%) are totally out of scope of our analysis, since our starting point is a changed production class. There is a similarly large set of commits that touches only production code. These are used in the association rule mining process, but contribute towards the `NONE` patterns. The large amount of production only commits can be the main reason of the weak links we found in the previous section. Finally, only 32% of commits are mixed, thus contain both production and test code changes. A proper aggregation of commits is a promising direction to avoid separated production and test changes.

TABLE VII
OVERVIEW OF SIZE AND COMPOSITION OF COMMITS

Project	Prod + Test commits	Avg size	Max size	Prod only	Mixed	Test only
commons-lang	1360	1.87132	65	532	428	395

We identified several threats to validity of this initial research. The difference between the history time frame considered may contribute towards the weak links (initial development phase vs stable development phase). We applied a source code based instrumentation in obtaining coverage links. The whole process depends on some thresholds (e.g. filtering) and we experienced uncertainty in specific steps of the original study. Despite these differences, we acknowledge

previous results and extend it with additional analysis. The analysis is production-code centric, due to the replication nature of the study; and we measured one system initially. These latter threats, together with the commit level analysis will be addressed in future work.

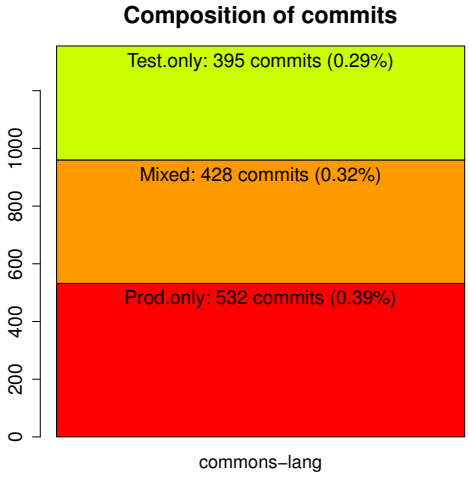

Fig. 9. Composition of commits

V. Conclusions

In this work association rule learning is applied to extract co-evolution patterns of production and test code. The analysis used fine-grained change data and dynamic coverage information of the project history to find out whether developers modify test code when they touch production code. We answer to the research questions as follows:

Answer to RQ1: Production and test code is developed in sync in the analyzed project. We provided detailed data on project history including coverage data. We investigated the composition of commits, where we found that in many cases production and test code are separately changed. Thus, aggregation of commits into logical units (like issues) could improve the co-evolution analysis.

Answer to RQ2: We successfully replicated the base study on the `commons-lang` project. We found weaker links than reported before – so we also provided analysis of change weighting – but we found that five of the six production to test code co-evolution patterns are present in the project history.

Several future research directions were identified during the analysis. There is an ongoing work in analyzing additional projects as well. Our main observation is that commit level separation may introduce bias in co-evolution analysis. We plan to introduce issue-level analysis in the near future.

Acknowledgment

This work was supported in part by the Research Council of the University of Klagenfurt and the Austria-Hungary Action Foundation (94öu8). László Vidács and Tibor Alex Szarka were supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

References

[1] Y. Jiang and B. Adams, "Co-evolution of infrastructure and source code: An empirical study," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 45–55.

[2] M. Langhammer, "Co-evolution of component-based architecture-model and object-oriented source code," in *Proceedings of the 18th International Doctoral Symposium on Components and Architecture*, ser. WCOP '13. New York, NY, USA: ACM, 2013, pp. 37–42.

[3] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 33:1–33:11.

[4] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley, "Recovering test-to-code traceability using slicing and textual analysis," *Journal of Systems and Software*, vol. 88, pp. 147–168, 2014.

[5] R. M. Parizi, S. P. Lee, and M. Dabbagh, "Achievements and Challenges in State-of-the-Art Software Traceability Between Test and Code Artifacts," *IEEE Trans. on Reliability*, vol. 63, no. 4, pp. 913–926, 2014.

[6] B. V. Rompaey and S. Demeyer, "Establishing traceability links between unit test cases and units under test," in *13th European Conference on Software Maintenance and Reengineering*, March 2009, pp. 209–218.

[7] A. Zaidman, B. V. Rompaey, S. Demeyer, and A. v. Deursen, "Mining software repositories to study co-evolution of production & test code," in *2008 1st International Conference on Software Testing, Verification, and Validation*, April 2008, pp. 220–229.

[8] C. Marsavina, D. Romano, and A. Zaidman, "Studying fine-grained co-evolution patterns of production and test code," in *Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 195–204.

[9] Z. Lubsen, A. Zaidman, and M. Pinzger, "Using association rules to study the co-evolution of production & test code," in *6th IEEE International Working Conference on Mining Software Repositories*, ser. MSR '09. IEEE Computer Society, 2009, pp. 151–154.

[10] L. S. Pinto, S. Sinha, and A. Orso, "Testevol: A tool for analyzing test-suite evolution," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 1303–1306.

[11] P. Marinescu, P. Hosek, and C. Cadar, "Covrig: A framework for the analysis of code, test, and coverage evolution in real software," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. ACM, 7 2014, pp. 93–104.

[12] M. Ghafari, C. Ghezzi, and K. Rubinov, "Automatically identifying focal methods under test in unit test cases," in *International Working Conference on Source Code Analysis and Manipulation (SCAM 2015)*. IEEE, sep 2015, pp. 61–70.

[13] A. Qusef, G. Bavota, R. Oliveto, A. D. Lucia, and D. Binkley, "Evaluating test-to-code traceability recovery methods through controlled experiments," *Journal of Software: Evolution and Process*, vol. 25, no. 11, pp. 1167–1191, nov 2013.

[14] A. Telea and D. Auber, "Code flows: Visualizing structural evolution of source code," *Computer Graphics Forum*, vol. 27, no. 3, pp. 831–838, 2008.

[15] B. Ens, D. Rea, R. Shpaner, H. Hemmati, J. E. Young, and P. Irani, "Chronotwigger: A visual analytics tool for understanding source and test co-evolution," in *2014 Second IEEE Working Conference on Software Visualization*, Sept 2014, pp. 117–126.

[16] D. Tengeri, Á. Beszédes, T. Gergely, L. Vidács, D. Havas, and T. Gyimóthy, "Beyond code coverage - an approach for test suite assessment and improvement," in *Intl Conference on Software Testing, Verification and Validation Workshops (TAIC PART 2015)*, Apr. 2015, pp. 1–7.

[17] C. Huo and J. Clause, "Interpreting Coverage Information Using Direct and Indirect Coverage," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, apr 2016, pp. 234–243.

[18] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, Nov 2007.

[19] F. Palomba, R. Oliveto, and A. D. Lucia, "Investigating code smell co-occurrences using association rule learning: A replicated study," in *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, Feb 2017, pp. 8–13.