TIPMerge: Recommending Developers for Merging Branches

Catarina Costa[1,2], Jair Figueiredo[1]
[1]Federal University of Acre
Rio Branco - AC, Brazil
{catarina,jjcfigueiredo}@ufac.br

Anita Sarma[3]
[3]Oregon State University
Corvallis, USA
anita.sarma@oregonstate.edu

Leonardo Murta[2]
[2]Fluminense Federal University
Niteroi – RJ, Brazil
leomurta@ic.uff.br

ABSTRACT
Development in large projects often involves branches, where changes are performed in parallel and merged periodically. This merge process often combines two independent and long sequences of commits that may have been performed by multiple, different developers. It is nontrivial to identify the right developer to perform the merge, as the developer must have enough understanding of changes in both branches to ensure that the merged changes comply with the objective of both lines of work (branches), which may have been active for several months. We designed and developed TIPMerge, a novel tool that recommends developers who are best suited to perform the merge between two given branches. TIPMerge does so by taking into consideration developers' past experience in the project, their changes in the branches, and the dependencies among modified files in the branches. In this paper we demonstrate TIPMerge over a real merge case from the Voldemort project.

CCS Concepts
Software and its engineering → Software configuration management and version control systems.

Keywords
Version Control, Branch Merge, Expertise Recommendation.

1. INTRODUCTION
The use of branching strategies is a common practice in collaborative software development to isolate new features from bug fixes, support customization, segregate development teams, etc. The task of integrating branches with parallel changes can be difficult [1], especially if multiple developers performed changes on the branches that need to be merged and if these changes conflict, directly or indirectly. For instance, we observed a merge case in the Rails project (https://goo.gl/7fP3fv), which included commits made by 47 developers in one branch, and 52 developers in the other branch [2, 3].

As branches can be active over long periods of time and involve changes from multiple developers, a single developer may not have the expertise necessary to merge all the changes in the branches. This sets the stage for a *collaborative merge* session. A collaborative merge can mitigate the missing expertise problem, as it brings together the parties that have the knowledge necessary to understand the reason behind the changes and resolve any conflicts that can occur [5]. In a recent survey [2], 75% of the developers said they need assistance to resolve conflicts. Ideally, the developers who are performing the collaborative merge have some knowledge about how to resolve conflicts during a merge.

Nevertheless, finding the right people to participate in a collaborative merge session is nontrivial. It requires knowledge about developers' contributions in each branch, understanding which files from one branch depends on the files in the other branch, and identifying who has expertise on the potentially conflicting files.

In this paper, we demonstrate TIPMerge[1][4], a tool that identifies the most appropriate developers to merge branches. For a given pair of branch, TIPMerge first identifies "key" files and the developers who have made changes to them in each branch. Key files are files that were changed in parallel across the branches, which may lead to direct conflicts; or files that have changed in one branch, but have dependencies with other changed files in the other branch, which may cause indirect conflicts. TIPMerge then identifies the overall experience of developers with the key files based on the branch and project history. After analyzing this information, TIPMerge recommends a ranked list of developers who are best suited to integrate a pair of branches.

We developed TIPMerge in java. Our current implementation is able to analyze Git repositories, independently of their programming language[2]. Our tool uses Git commands to extract the repository information, and Dominoes [7–9] to identify logical dependencies among files. After identifying the necessary information, TIPMerge uses a medal counting system that checks contributions in key files to rank the most appropriate developers to perform the merge.

In our previous work [4], we have discussed the evaluation of TIPMerge, which included a quantitative analysis of 28 real-world projects, which included up to 15,584 merges with at least two developers, and qualitative analysis of two projects to better understand TIPMerge recommendations. We found that in 85% of the top-3 recommendations, we included the developer who actually performed the merge. In this paper, we demonstrate how TIPMerge is designed to be used by an end-user. We use a real merge case from the Voldemort project[3] to show case TIPMerge. We also provide complementary material (videos, virtual machines, etc.) in the wiki of the project website[4].

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2983936

[1] TIPMerge is available at https://github.com/gems-uff/tipmerge as an open sourced tool (MIT License).
[2] TIPMerge is language agnostic when analyzing expertise at the file-level. At the method-level, it currently analyzes Java projects only.
[3] https://github.com/voldemort/voldemort.
[4] https://github.com/gems-uff/tipmerge/wiki.

2. SCENARIO

We first present a real world, albeit small, scenario to illustrate the use of branches. We selected a merge case in the Voldemort project with a reasonable number of commits to explain how our tool works. This scenario is the merge that occurred on August 9, 2014 between the two branches: coordinator_admin (Branch1) and coadmin (Branch2). The last commit in coordinator_admin was 5aabfbe and in coadmin was c5995a7. Two developers (Felix and Siddharth) performed seven commits over 12 files in the coordinator_admin branch. On the other hand, five developers (Arunachalam, Felix, Siddharth, Xu, and Zhongjie) performed 33 commits over 45 files in the coadmin branch. All the 12 files changed in coordinator_admin branch were also changed in the coadmin branch. It means that these files might have generated direct conflicts if they involved overlapping changes. Note, we consider a direct (merge) conflict to occur if the git-merge command fails. Moreover, these 12 files may also have dependencies with any of the 45 files, potentially leading to indirect conflicts.

Finally, the previous history of the project consists of all the commits from the beginning of the project until the moment when these branches were forked. During this period, the following 27 developers worked over the 45 files: Alex, Abhinay, Arunachalam, Baepiff, Bhavani, Bhupesh, Chinmay, David, Elias, Felix, Geir, Holden, Ismael, Jakob, James, Jay, Karthik, Kirk, Lei, Michael, Peter, Roshan, Siddharth, Vinoth, Xu, Yair, and Zhongjie.

We use this scenario to demonstrate how TIPMerge can answer the question "*Who are the most appropriate developers to perform this merge?*"

3. TIPMERGE IN ACTION

In this section, we demonstrate how TIPMerge can be used to recommend the developer who is the most appropriate for merging two given branches. Figure 1 depicts the TIPMerge user interface. Once the user has selected a project (Figure 1(a), voldemort), TIPMerge presents information about the project, along with details about the merges and branches (Figure 1(b)), and a histogram of the commits by developers, ordered from highest number of commits to lowest (Figure 1(c)).

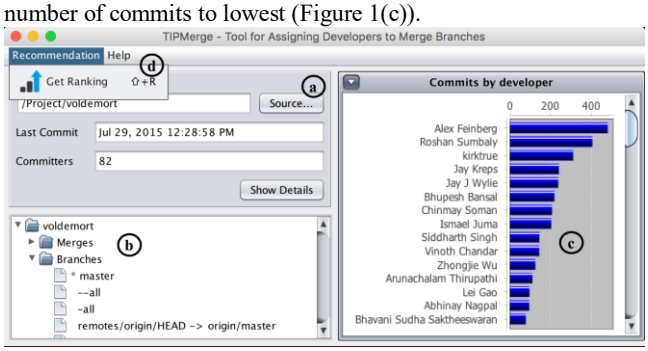


Figure 1. TIPMerge Interface

A user can request a recommendation about the most appropriate developers to merge branches by clicking the *Recommendation - Get Ranking* button (Figure 1 (d)), which takes them to the Recommendation menu (Figure 2). Here the user selects the branches that they are interested in merging. They can use the drop down menu to select the branch name and the specific commit (hash) in a branch that they desire to merge. Here we see that the coordinator_admin and the coadmin branch are selected. Note that if the hash is left empty, then the last commit at each branch tip is automatically selected.

TIPMerge then executes the following steps to provide a recommendation (explained in the ensuing subsections):

1. Extract data from the repository until the branches tips. That is, the two most recent commits of the two branches that will be merged.
2. Detect dependencies among files by identifying files that were frequently co-committed (logical coupling). We calculate dependencies from the data before the branches forked.
3. Identify developers who edited key files. We collect this information both for changes in the branches, and for changes in the previous history.
4. Recommend a ranked list of suitable candidates to perform the merge based on a medal counting system.

3.1 Data Extraction

The first step in the process is extracting the change data from the branches. To do so, we first need the branches that will be merged. The end user selects the branches by using the TIPMerge UI, as shown in Figure 2. They first select the two branch names (Figure 2(a)) and the two hashes to merge (Figure 2(b)). For instance, in our guiding example, coordinator_admin (Branch1) and coadmin (Branch2) are the two branches, and commits 5aabfbe… and c5995a7… are the commit at the tips of these branches that are to be merged. From these commits it is possible to identify all commits that belong to each branch, which comprises all commits between each of the branch tips to the common ancestor (commit c07b777). Additionally, we extract information from the first commit until the common ancestor to identify expertise of developers in the previous history. We do not extract information of files changed in merge commits. This decision was taken to prevent TIPMerge accounting merged files to the developer responsible for merging. In many cases, this developer just automatically combines two versions, not in fact editing the files. Furthermore, should we consider the previous merge commits, we could bias our approach, since this could increase the changes of indicating the same developer in the future.

A user can trigger the recommendation analysis by clicking on the *Run* button (Figure 2(c)). Users can also filter file extensions that they know are irrelevant to the recommendation. An obvious example can be a Readme.txt file. Once TIPMerge analyzes the project information, it presents the files that each developer has edited, and the edit frequency in terms of commits (Figure 2(d)). This information is provided for files changed individually in each branch, together in both branches, and in the previous history.

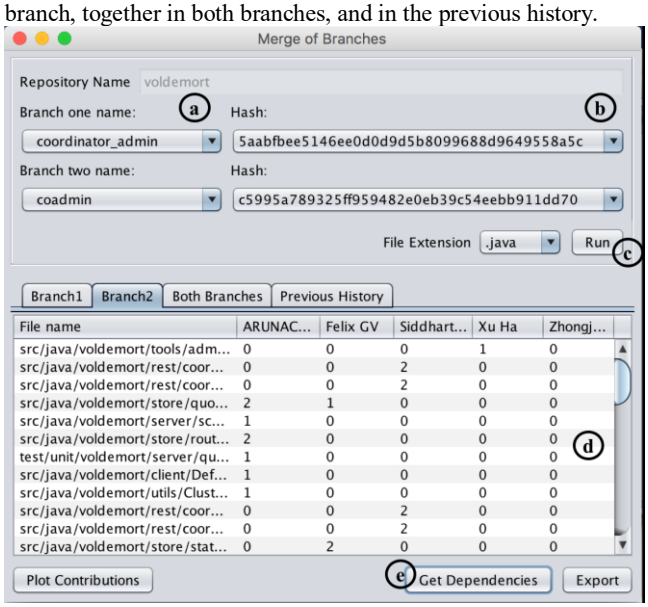


Figure 2. Information about branches and previous history

3.2 Dependency Detection

Next, we identify dependencies among files that were edited across branches by using the concept of logical coupling. Logical coupling detects evolutionary dependencies by identifying files that are frequently changed together [6, 10], and is programming language agnostic. In open source projects, the use of logical coupling is the most efficient as different projects use different programming languages, and several projects use a combination of different programming languages.

We adapted the Dominoes [7–9] tool to identify logical dependencies among files. Dominoes is a library developed in java, which processes information extracted from Git repositories and stores them in a SQLite database. Dominoes organizes data extracted from software repositories into matrices to denote relationships among software entities. For example, $[commit|file]$ denotes the files that were changed by commits in the project. These matrices are then combined using GPU, for performance efficiency, to depict higher-order relationships, such as logical dependencies among files: $[file|file] = [commit|file]^T \times [commit|file]$.

We extended the data extraction part of Dominoes to allow the extraction and identification of information about commits that are distributed across branches.

We use the edit history of the project (before the branching occurred) to determine dependencies between pairs of files. The past history provides us a baseline of these dependencies. However, we only consider files changed across branches during this analysis. This is vital, since only these files are subject to indirect conflicts when the branches are merged.

Dominoes provides us with some basic (data mining) metrics, such as confidence, when computing the logical dependencies among files. Confidence values range from 0 to 1, where a value of 1 means that every time a file is changed, the other file is also changed. In this case, the use of a threshold is necessary because low confidence implies low probability that changing a file causes impact in the dependent file. As confidence is directional, dependencies are not symmetric; that is file A may depend on file B, without requiring file B dependency on file A. Development teams have the freedom to decide the threshold above which a dependency becomes relevant.

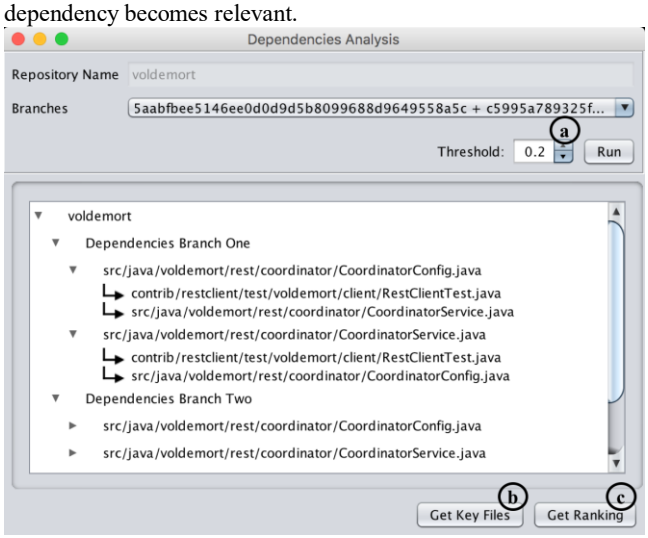


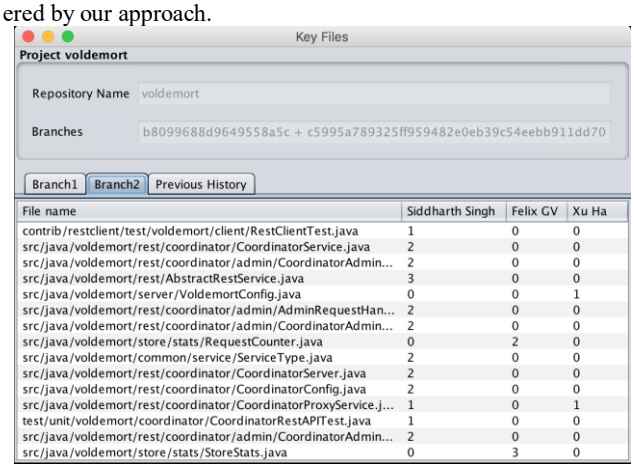
Figure 3. File Dependencies

The user can check the logical dependencies (Figure 2(e)) by clicking on the *Get Dependencies* button, which open the *Dependencies Analysis* window (Figure 3). Here, the user can configure the confidence threshold to visualize the logical dependencies (Figure 3(a)). In our (simple) scenario, there are many dependencies (19 dependency relationships) when we use a threshold of 0.2. Note, since we explicitly chose a small example we had to use a low threshold. Our quantitative evaluation in prior work used a threshold of 0.6.

3.3 Key File Author Identification

The next step in our approach is to identify the developers who have modified files that are relevant to the merging of the selected branches. We term these files as key files. These are files that have been changed in parallel in both branches, potentially leading to direct conflicts; or that were changed in one branch, but have dependencies with files that were changed in the other branch, potentially leading to indirect conflicts. In the latter case, both the dependent and dependency files are considered key files. Only key files are relevant for us, as all other files can be automatically merged safely. Files that were not changed in either branch are irrelevant for the merge. A user can click on the *Get Key Files* button (Figure 3(b)), to see who changed the key files (Figure 4).

In our scenario, 12 files were changed in both branches. Of the 19 files that had dependencies across branches, only 3 were not changed in both branches. So, 15 unique key files were identified (Figure 4). It is important to note that although 5 developers had changed files in coadmin (Figure 2), only 3 developers changed key files: Felix, Siddiharth, and Xu. On the other hand, all developers that worked on coordinator_admin altered key files (Felix and Siddharth). From the 27 developers that worked in the previous history, 21 changed key files. All developers who have knowledge about key files in the branches or history are considered by our approach.



File name	Siddharth Singh	Felix GV	Xu Ha
contrib/restclient/test/voldemort/client/RestClientTest.java	1	0	0
src/java/voldemort/rest/coordinator/CoordinatorService.java	2	0	0
src/java/voldemort/rest/coordinator/admin/CoordinatorAdmin...	2	0	0
src/java/voldemort/rest/AbstractRestService.java	3	0	0
src/java/voldemort/server/VoldemortConfig.java	0	0	1
src/java/voldemort/rest/coordinator/admin/AdminRequestHan...	2	0	0
src/java/voldemort/rest/coordinator/admin/CoordinatorAdmin...	2	0	0
src/java/voldemort/store/stats/RequestCounter.java	0	2	0
src/java/voldemort/common/service/ServiceType.java	2	0	0
src/java/voldemort/rest/coordinator/CoordinatorServer.java	2	0	0
src/java/voldemort/rest/coordinator/CoordinatorConfig.java	2	0	0
src/java/voldemort/rest/coordinator/CoordinatorProxyService.j...	1	0	1
test/unit/voldemort/coordinator/CoordinatorRestAPITest.java	1	0	0
src/java/voldemort/rest/coordinator/admin/CoordinatorAdmin...	2	0	0
src/java/voldemort/store/stats/StoreStats.java	0	3	0

Figure 4. Key Files

Once we have identified the key files and the developers that have experience with them, we are able to calculate the developer recommendations.

3.4 Developer Recommendation

After identifying key files and the developers that have changed such files, TIPMerge applies an algorithm to quantify their contributions by considering whether the files were changed in a branch or in the previous history. We use a medal system to determine the position of each developer in the final recommendation ranking. This is analogous to how countries are ranked in the Olympic Games based on medal counts. The following rules define when developers receive gold, silver, and bronze medals.

A *gold medal* is awarded when a developer changes a key file in a branch. The rationale is that the developer who changed a key file is the most knowledgeable about the change and its implications. They probably are also well versed with the file in general, and, therefore, likely to be able to perform additional edits during

a merge if necessary. In our scenario, all those developers who edited any of the 12 files that were changed in both branches, or any of the 19 files involved in a dependency relationship receive a gold medal for each changed (key) file. Note, developers can receive a maximum of two gold medals for a key file, if they changed the same file in both branches.

A *silver medal* is awarded when a developer has changed a key file in the past. Developers who created or edited files in the past likely possess knowledge about the goals and requirements of these files, which can be helpful when performing a merge. In our scenario, the 21 developers who changed key files in the previous history receive a silver medal for each edited (key) file.

A *bronze medal* is awarded when a developer changes a file that depends on another file. Both the dependent and the dependency are key files, but only who changed the dependent file receives a bronze medal. The logic is that developers who have changed a dependent file, may have learned about the API of the file that they are using. Consequently, they may know the goals and expectations of such a file, which may help in determining the impact of a change. In our example, Felix received a bronze medal because he changed RequestCounter.java in coadmin, and this file depends on file CoordinatorService.java changed in coordinator_admin.

We assign a medal for each file that was edited, irrespective of the number of commits made to the file. In our approach, we assume that when a developer edits a file, that developer has knowledge about the entire file. While our approach can support a finer-grained expertise calculation at the method level, we leave it for future work.

Our algorithm prioritizes developers with gold medals, because: (1) they are the expert on the change that has been made, and (2) they have the most recent knowledge about the file to which a change has been made.

In the case of a tie in gold medals, we use the number of silver medals to break the tie. This is because, everything being equal, a developer who has more experience overall is likely to be more suitable in merging changes. Similarly, in the case of a tie in silver medals, we consider bronze medals. The notion is that if two developers have equal number of changes that they have made, and equal knowledge about the project history, then a developer who has additional knowledge about another file is more appropriate.

Using this medal count system, TIPMerge generates a ranking of suitable candidates to perform the merge between a pair of branches. The user can access the *recommendation ranking* by clicking on the *Ranking* button (Figure 3(c)), which leads them to the developer recommendation page (Figure 5) For each developer (Figure 5(a)) and each file ((Figure 5(b)), TIPMerge lists the number of gold, silver, and bronze medals.

TIPMerge also shows the branch in which a change was made ((Figure 5(c)). In this case, we expand the changes made by Felix. Changes are denoted by the black dot on the branch icon. The straight "branch line" denotes branch one (here, coordinator_admin) and the angular "branch line" denotes branch two (here, coadmin). We see that Felix received three gold medals, because of his changes to 3 of the 4 files listed in Figure 5(b). Further, we note that he made changes to both braches, twice to coordinator_admin and once to coadmin. Similarly, he received three silver files on account of his changes to the previous history (denoted by the dot on the line before the branches forked).

Felix received a bronze medal because he changed two files in Branch2 - coadmin (RequestCounter.java and StoreStats.java) that depends on file CoordinatorService.java, changed in Branch1-coordinator_admin. This is denoted by the arrow extending from the changes in Branch2 to the files in Branch1. Felix gets a bronze medal as we assume that Felix knows about the file CoordinatorService.java, because he has changed files that call this file.

We see that Siddarth is at the top of our recommendation on account of the high number of gold medals that he received. We note that Felix comes second in our recommendation, which we explore further. He has changed 3 files in the branches (1 in coordinator_admin and 2 in coadmin), 3 files in the history, and changed a file in coadmin that depends on a file in coordinator_admin. In fact, he was actually the developer in charge of merging these two branches (commit 8358888).

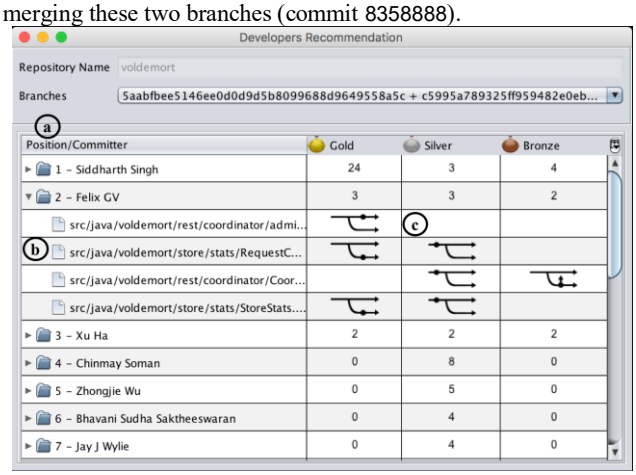


Figure 5. Recommendation ranks for Voldemort project

We evaluated TIPMerge by running our analysis on 28 software projects. On average, 85% of the top-3 recommendations by TIPMerge correctly included the developer who actually performed the merge. Moreover, in 82% of the merges, TIPMerge obtained a higher accuracy than selecting the developer who performed most of the previous merges (i.e., the majority class). We further investigated the cases where TIPMerge recommendations were incorrect by interviewing developers from two projects. In many cases, interviewees agreed that the TIPMerge recommendations were accurate. And in some cases, we found that the recommended developer had actually participated in a collaborative merge. Details about the evaluation are in our prior work [4].

4. CONCLUSIONS

In this paper, we demonstrate how TIPMerge can be used through an example of a real merge case from the Voldemort project. We explain how TIPMerge creates its recommendation by analyzing the contributions of developers in branches, as well as the previous history, and by taking into consideration both potential direct or indirect conflicts that might arise during the merge.

Our results indicate that TIPMerge is helpful in identifying the developers who have the most knowledge about key files and therefore, have the expertise to perform the merge. We also believe that our approach can be helpful to enable the lead integrator find developers for collaborative merges or help in integration.

Further, we plan to run the analysis at a finer grain (method level), as this would provide a detailed understanding of file dependencies and developer knowledge about specific parts of the code base. We also plan to verify developers with complementary expertise to conduct collaborative merge sessions.

5. ACKNOWLEDGMENTS

This work is partially supported by CAPES (10614-14-1), CNPq, FAPERJ, NSF CCF-1253786 and HCC- 1559657.

6. REFERENCES

[1] Bird, C. and Zimmermann, T. 2012. Assessing the Value of Branches with What-if Analysis. *ACM SIGSOFT Int'l Symp. Foundations of Software Eng (FSE)* (New York, NY, USA, 2012), 45:1–45:11.

[2] Costa, C., Figueiredo, J.J.C., Ghiotto, G. and Murta, L. 2014. Characterizing the Problem of Developers' Assignment for Merging Branches. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*. 24, 10 (Dec. 2014), 1489–1508.

[3] Costa, C., Figueirêdo, J.J.C. and Murta, L. 2014. Collaborative Merge in Distributed Software Development: Who Should Participate? *The International Conference on Software Engineering and Knowledge Engineering (SEKE)* (Vancouver, Canada, 2014), 268–273.

[4] Costa, C., Figueiredo, J., Murta, L. and Sarma, A. 2016. TIP-Merge: Recommending Experts for Integrating Changes across Branches. *ACM SIGSOFT Int'l Symp. Foundations of Software Eng. (FSE)* (Seattle, WA, USA, 2016).

[5] Nieminen, A. 2012. Real-time collaborative resolving of merge conflicts. *2012 8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)* (2012), 540–543.

[6] Oliva, G.A. and Gerosa, M.A. 2011. On the Interplay between Structural and Logical Dependencies in Open-Source Software. *2011 25th Brazilian Symposium on Software Engineering (SBES)* (Sep. 2011), 144–153.

[7] Da Silva, J.R., Clua, E., Murta, L. and Sarma, A. 2015. Multi-Perspective Exploratory Analysis of Software Development Data. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*. 25, 01 (Feb. 2015), 51–68.

[8] Da Silva, J.R., Clua, E., Murta, L. and Sarma, A. 2015. Niche vs. breadth: Calculating expertise over time through a fine-grained analysis. *2015 IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)* (Mar. 2015), 409–418.

[9] Da Silva Junior, J.R., Clua, E., Murta, L. and Sarma, A. 2014. Exploratory Data Analysis of Software Repositories via GPU Processing. *The International Conference on Software Engineering and Knowledge Engineering (SEKE)* (Vancouver, Canada, 2014).

[10] Zimmermann, T., Weisgerber, P., Diehl, S. and Zeller, A. 2004. Mining Version Histories to Guide Software Changes. *Proceedings of the 26th International Conference on Software Engineering (ICSE)* (Washington, DC, USA, 2004), 563–572.