Assessing Semistructured Merge in Version Control Systems: A Replicated Experiment

Guilherme Cavalcanti, Paola Accioly and Paulo Borba
Federal University of Pernambuco
Recife, Brazil
{gjcc,prga,phmb}@cin.ufpe.br

***Abstract*— *Context*: To reduce the integration effort arising from conflicting changes resulting from collaborative software development tasks, unstructured merge tools try to automatically solve part of the conflicts via textual similarity, whereas structured and semistructured merge tools try to go further by exploiting the syntactic structure of the involved artefacts. *Objective*: In this paper, aiming at increasing the existing body of evidence and assessing results for systems developed under an alternative version control paradigm, we replicate an experiment conducted by Apel et al. [1] to compare the unstructured and semistructured approach with respect to the occurrence of conflicts reported by both approaches. *Method*: We used both semistructured and unstructured merge in a sample 2.5 times bigger than the original study regarding the number of projects and 18 times bigger regarding the number of merge scenarios, and we compared the occurrence of conflicts. *Results*: Similar to the original study, we observed that semistructured merge reduces the number of conflicts in 55% of the scenarios of the new sample. However, the observed average conflict reduction of 62% in these scenarios is far superior than what has been observed before. We also bring new evidence that the use of semistructured merge can reduce the occurrence of conflicting merge scenarios by half. *Conclusions*: Our findings reinforce the benefits of exploiting the syntactic structure of the artefacts involved in code integration. Besides, the reductions observed in the number and size of conflicts suggest that the use of semistructured merge, when compared to the unstructured approach, might decrease integration effort without compromising correctness.**

Keywords*— *replication study, collaborative development, software merging, semistructured merge, version control systems.

I. Introduction

In a collaborative development environment, developers often implement tasks in an independent way using individual copies of project files. As a result, while merging separate code contributions from each task, one likely has to deal with conflicting changes and dedicate substantial effort to resolve conflicts. These conflicts occur due to a number of reasons. For example, when different developers make changes to the same artefact without being aware of the other changes — the so-called *direct* or *textual* conflicts — or when there are concurrent modifications in different artefacts, leading to build or test failures — the *indirect* conflicts [2, 3]. Regardless of the nature of the conflicts, they may hamper productivity, since detecting and solving conflicts might be a tiresome and error prone activity, and, as a consequence, they delay the project while developers trace its cause and seek a solution.

To learn about the occurrence of conflicts and their consequences, previous empirical studies answer questions concerning when developers detect conflicts, and how often conflicts occur. Zimmermann [4], for instance, describes that textual conflicts occurred in a range from 23% to 47% of all files' integration. Brun et al. [2] and Kasi and Sarma [5] found that textual conflicts occurred in an average of 15% of all merge scenarios — a set consisting of a common base revision and its derived versions — and 31% of the merge scenarios free of textual conflicts resulted in build or behavioral errors.

Such evidence motivates and guides the design of tools that use different strategies to both decrease integration effort and improve correctness during code integration. For example, to reduce the integration effort, unstructured merge tools are purely text-based and resolve conflicts via textual similarity. On the other hand, a structured merge tool is tailored to a specific programming language and uses knowledge of the language's grammar to resolve conflicts [6, 7, 8, 9, 10]. Finally, semistructured merge [1] attempts to combine the previous ones, so that it provides structural information about software artefacts to resolve conflicts automatically, and when this information is not sufficient, it applies the usual textual resolution to the conflict.

Apel et al. [1] in a previous empirical study found that the semistructured approach was promising if compared to the unstructured one. By studying 24 projects using Subversion, a Centralized Version Control System (CVCS), and analysing a total of 180 merge scenarios, they found that, in 60% of their sample merge scenarios, the semistructured approach was able to reduce the number of textual conflicts by, on average, 34%. They also found that, in 82% of their sample merge scenarios, the semistructured approach reduced the number of conflicting lines of code by, on average, 61%; and that, in 72% of their sample merge scenarios, semistructured merge reduced the number of conflicting files by, on average, 28%.

Given the importance of such tools for collaborative software development, here we further investigate Apel's et al. [1] hypothesis and replicate their study. To possibly expand external validity of the original study, we analyse different systems stored on Git, a Distributed Version Control System (DVCS), since DVCSs have seen an increase in popularity compared to traditional CVCS, and offer extra information that help us to better understand software development processes, such as merge tracking [11, 12]. We then compare semistructured merge to the unstructured one in 3266 merge scenarios from 60 projects, a sample 2.5 times bigger than the original study regarding the number of projects and 18 times bigger regarding the number of merge scenarios. Besides, our methodology allows us to collect evidence about the occurrence of conflicting code integrations and, thus, compare with those from the studies of Brun et al. [2] and Kasi and Sarma [5]. While this evidence rely on the use of

an unstructured merge tool, here, we are able to bring a new evidence about the occurrence of conflicting code integrations considering the use of a semistructured merge implementation.

In the remainder of the paper, we provide a more detailed background of the different merge approaches (Section II), followed by the description of the replication study design (Section III), the evaluation results (Section IV) and discussion on the results (Section V). Finally, we present the threats to validity in Section VI, and our final considerations in Section VII.

All the data and scripts used in our study are publicly available online.[1]

II. Merge Approaches

In the context of collaborative development, VCSs play a fundamental role because they allow developers to work independently on their tasks, using individual copies of project files. With the use of VCSs developers create a revision of the software system from a base version, develop and evolve it separately, and finally integrate it back into the base version in a process called *three-way merge*, which is used in every practical version control system. A three-way merge aims at joining two independently developed versions based on their common ancestor (the version from which both have been derived) by locating their differences, selecting and applying corresponding changes to the merged version [13].

However, during the three-way merge process, one likely has to deal with conflicting changes and dedicate substantial effort to resolve conflicts using some merge tool. To reduce such effort, while unstructured merge tools try to automatically solve part of the conflicts via textual similarity, structured and semistructured merge tools try to go further by exploiting the syntactic structure of the artefacts involved, this way they reduce spurious conflicts due to changes, for example, in the order of commutative and associative declarations.

To better understand the nature of the different merge tools, consider the snippets in Figure 1 showing that the tasks *Authentication* and *Research Group* are each assigned to a different developer, and both have to edit the *Member* class. The developer responsible for the *Authentication* task adds a new field declaration representing the member's username, and the developer responsible for the *Research Group* task adds a new field declaration representing the member's email.

By using a **unstructured merge** tool, such as `diff` and `merge` of Unix, used in version control systems such as CVS, Subversion, and Git, the code integration of Figure 1 would result in a conflict. This happens because unstructured merge operates purely on plain text or tokens, and if two revisions modify or extend the text in the same part of the code, the tool notifies a conflict [14]. That is, it is not able to decide how to merge the modifications or extensions. In the example, the unstructured merge does not detect that these extensions are field declarations and that their order does not matter within the class to which they belong. If the merge tool was able to detect this condition, it would be able to resolve the conflict automatically since it might include one of the field declarations first and then the other after, and vice versa.

In turn, **structured merge** improves the unstructured approach with regard to conflict detection and resolution by exploiting the artefacts' structure. It is specific to a programming language and uses information inherent to

[1] http://goo.gl/hCrI3H

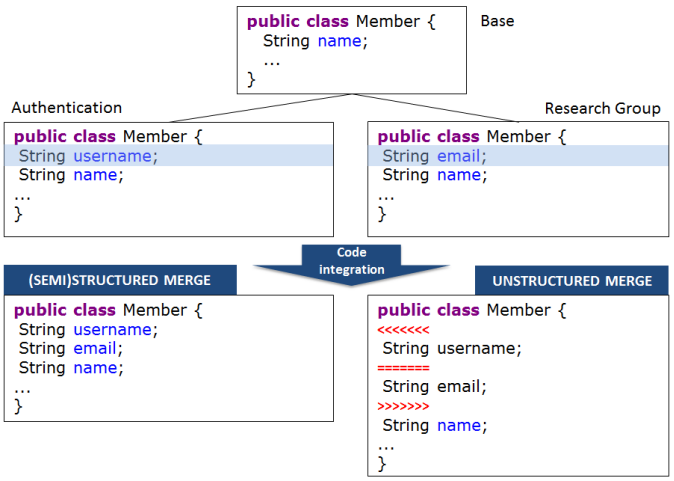

Fig. 1. Developing tasks independently can lead to conflicting changes.

the artefacts of the language to solve the largest possible number of conflicts automatically. There are implementations using structural information such as the context-free and context-sensitive syntax during the merge process [8, 15], or even representing software artefacts as graphs and trees in tools for Java [6, 9] and C++ [7].

Finally, **semistructured merge** represents software artefacts as trees and provides information (through an annotated grammar) about how nodes of certain types (methods, classes, etc.) and its subtrees can be merged (via superimposition [16], which merge trees recursively, beginning from the root, based on structural and nominal similarities). Apel et al. [1] state that the ability of semistructured merge to resolve certain conflicts is based on the observation that the order of certain elements (classes, methods, fields, imports, and so on) does not matter – the so-called *ordering conflicts*. By abstracting the document structure as a tree, semistructured merge has enough information to identify ordered items. Besides, when semistructured merge cannot merge a software element, such as method bodies with statements, it represents the elements as plain text and uses the conventional unstructured merge. It also allows special conflict handlers to be added, to resolve specific cases of textual conflicts, such as conflicts due to concurrent modifications to access modifiers, implements list and so on. Thus, the approach becomes a combination of the unstructured and structured approaches. The semistructured merge is more expressive than the unstructured approach because it resolves conflicts automatically based on the information available about the language; and more general than the structured one, since it supports a larger number of languages by providing an annotated grammar of the language to be supported.

So, by using both a semistructured or structured merge tool, the code integration of Figure 1 would not report conflicts.

III. Replication Study Design

Since semistructured merge is able to resolve conflicts that unstructured merge cannot resolve, such as the ordering conflicts, we expect that the semistructured approach is able to decrease the occurrence of conflicts compared with the unstructured one. In this case, it is interesting to know how big is the reduction, and how frequently such conflicts occur in real software projects. This is what led Apel et al. [1] to

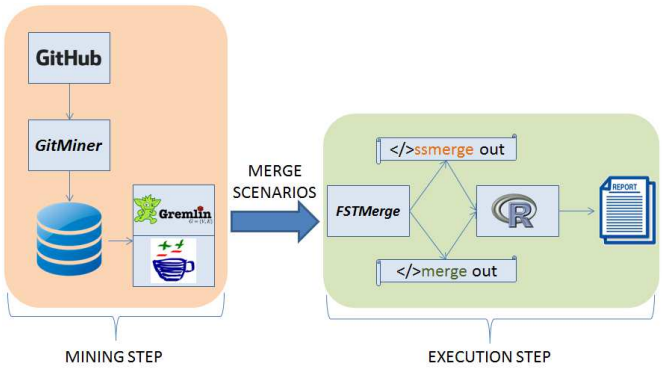

Fig. 2. Replication Study Design.

conduct an empirical study to evaluate semistructured merge.

To further evaluate semistructured merge in a different context formed by different systems and version control paradigm, as a replicated study, we want to investigate the original study hypothesis:

Original Hypothesis. *Many of the conflicts that occur in merging revisions are ordering conflicts, which can be resolved automatically with semistructured merge. An additional fraction of conflicts can be resolved with conflict handlers.*

Besides, we want to go further and provide evidence of the occurrence of conflicting merge scenarios as done in the studies of Brun et al. [2] and Kasi and Sarma [5]. While this evidence rely on the use of a unstrucutured merge tool, we aim to provide a new evidence based on the use of the semistructured approach.

To this end, we designed a two-step study as illustrated in Figure 2. The study design is composed by a *mining* step, which is different from the original study since we are exploring DVCS repositories instead of CVCS ones; and by a *execution* step, which is similar to the original study, because we use the tool and scripts provided by the original authors. In particular, in the mining step, we built tools that mine DVCS repositories to collect a number of merge scenarios. Subsequently, in the execution step, we use a prototype of the semistructured approach (`FSTMerge`) in order to run the selected merge scenarios using both merge approaches, and R scripts, to collect metrics on the number of conflict headers per file (*textual conflicts*), lines of code surrounded by those conflict headers (*conflicting lines of code*) and files with at least one conflict header (*conflicting files*).

In addition, to assess the second part of the hypothesis, both studies evaluate the occurrence of conflicts that could be resolved with special conflict handlers — also called *semantic* conflicts in the original study.

Finally, to learn more about the nature of the studied merge approaches, we reviewed some of the merged revisions of the projects manually.

A. Mining Step

In the same way as the original study, we analyse the source code history of real projects instead of developing our own case study, which could leave too much room for bias. More importantly, how to select merge and conflicting scenarios is fundamental to the study since we want to expand the external validity of the original study.

For mining's purpose, Apel et al. [1] explored the SourceForge open-source software portal based on two criteria. First, the projects must be of reasonable but varying sizes. Second, either semistructured merge or unstructured merge must produce at least one conflict. They also analysed the project's log to extract information about the merges that developers actually performed and the revisions involved, and merges that could have been performed or that are realistic considering the revision history. The former was indicated by comments of the developers that point clearly to merges, and the second by patterns indicating sequence of multiple, alternating changes in different branches (e.g., trunk-branch-trunk), which indicates concurrent development and points to potential conflict scenarios (as long as the changes in different branches are not identical). Technically, they used Subversion to browse the revision histories and to check out revisions. Finally, they selected a sample of 24 projects with 180 merge scenarios written in Java, Python and C#.

We decided to explore Git and GitHub because they offer extra information that help us to better understand project's development processes, including merge tracking, which allows us to conduct our mining step in an automatic and systematic fashion.

We chose projects candidates based on three criteria. First, *number of commits*, because we believe that the greater the number of commits, the greater is the possibility of finding a merge commit — a commit that represents a three-way merge. Second, *number of developers*, because we believe that the greater the number of developers, the greater is the possibility of having conflicting code contributions resulting from developers' tasks, although only a portion of them may actually have worked on the project. Third, *the frequency and recency of its developers activities*, because we do not want inactive projects, which could denote that they are toys or projects that do not reflect any current model of development. The first and second criteria were based on information from the projects' repository pages on GitHub, and the last, on GitHub's Trending[2], which indicates projects that were more active and popular at the time of the study. Regarding the sampling of merge scenarios, we were interested on those which had already shown conflicts on Git to provide the evidence of the occurrence of conflicting merge scenarios. Therefore, we used tools to mine GitHub and to reproduce Git merges.

We mined GitHub using the `GitMiner`[3] tool. `GitMiner` receives a project or a particular GitHub user, connects to GitHub via GitHub's API and loads all the information available from the project or user and copies of the repositories' source code. Finally, the data is stored in a Neo4j[4] graph database. Such architecture is the most suitable for understanding individual dependencies between projects and its users activities, including their commits [17], which is fundamental in our mining step.

We then built a script to query the database to retrieve users commits and to execute code integration using Git built-in unstructured merge tool. Our script uses `Gremlin`[5], a graph traversal language, and `JGit`[6], a Java library implementing the Git version control system. The graph

[2]https://github.com/trending
[3]https://github.com/pridkett/gitminer
[4]http://www.neo4j.org/
[5]https://github.com/tinkerpop/gremlin/
[6]http://www.eclipse.org/jgit/

database represents commits as nodes with an `isMerge` attribute indicating whether the commit is a merge commit. In addition, each merge commit has two parents — here we call them left and right revisions — with a common ancestor — the base revision. Therefore, to identify merge scenarios, we query (1) the ID of all merge commits, checking which commits have the `isMerge` attribute with a true value, and (2) the ID of the revisions (base, left and right) that lead to the merge commit, and, thus, constitute the merge scenario. Finally, since we are interested on merge scenarios that had already shown conflicts, we use Git built-in unstructured merge tool to merge the revisions from the merge scenarios and we filter them by the conflicting ones, looking for those which have at least one file marked with conflict headers.

Following this procedure, we came to a sample of 4678 merge scenarios from 60 projects, written in Java, Python and C#. However, unfortunately, we had to discard 1412 merge scenarios because the semistructured implementation used in both studies does not support all these scenarios due to incompatibility between its current annotated grammar and the source code of the merge scenario. For example, there is an incompatibility with object initializers with named objects in C# and conditional assignment in Python, leading to parser's errors. As we will discuss in Section VI, this issue might threat internal validity because we could bias our results with scenarios only supported by the semistructured merge tool used in the study. We therefore run the study with 3266 merge scenarios, a number 18 times bigger than the original study (and 2.5 times bigger regarding the number of projects), which possibly expands the external validity of the original study. (The original study does not mention any incompatibility issue and discards.)

As we will discuss on Section VI, it is important to note that, despite our assumptions underlying our project selection criteria, the development model adopted by the projects as well as Git internal mechanisms — such as `git rebase`, which allows the history of the repository to be rewritten — might decrease the occurrence of merge commits and conflicting merges commits. More specifically, we analyse projects that adopt a pull-based development, therefore, we may have detected fewer conflicts that could exist if the project used a push-based development because in such cases the conflict may have been perceived by the integrator, who may have rejected the pull. We may also have detected less conflicts because merge commits that led to conflicts may have been erased from project's history with `git rebase`. For instance, as shown in Figure 3, *django* is the project with the largest number of collaborators, a large number of commits, but proportionately few merge commits and conflicting merge commits. Its documentation[7] makes explicit that they use a pull-based development model, with the frequent use of `git rebase` in the contribution process. In turn, *cassandra* has one eleventh of django's collaborators, less commits, but nine times more merge commits and far more conflicting merge commits. This happens because this project has a patch-based contribution process, with no specific strategy to avoid conflicts.[8](All information on the sample is available on our online appendix.)

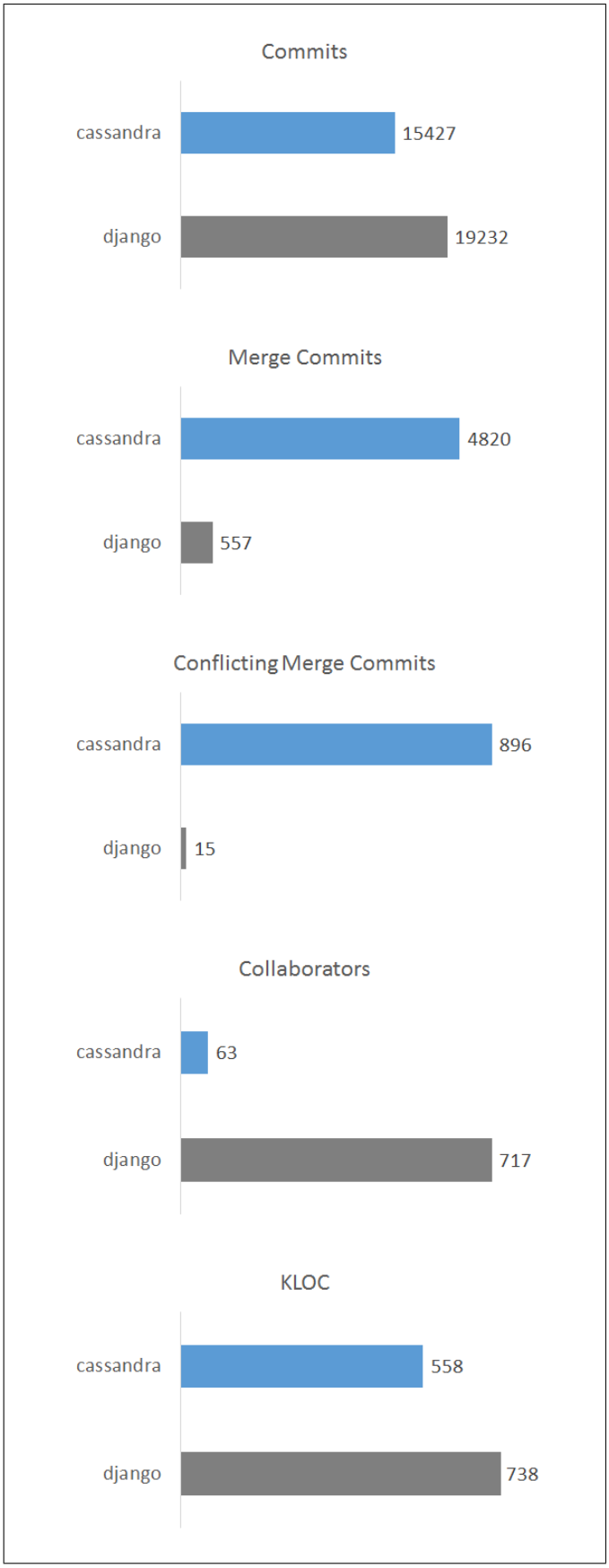

Fig. 3. Projects django and cassandra characteristics.

[7]https://docs.djangoproject.com/en/1.7/internals/contributing/writing-code/working-with-git/

[8]http://wiki.apache.org/cassandra/HowToContribute

B. Execution Step

After collecting the sample projects and merge scenarios in the previous step, we apply both unstructured and semistructured approaches to the merge scenarios and we analyse the number of textual conflicts, conflicting lines of code and conflicting files resulting from both unstructured and semistructured merge. Besides that, we also count the occurrence of semantic conflicts. We performed this step exactly like the original study, using the same tools and scripts.

In the original study, the authors implemented a first prototype of a semistructured merge tool, called `FSTMERGE`, which is able to resolve ordering conflicts and which can be extended with special conflict handlers. Currently, the tool has conflict handlers for 54 structural elements of Java, C#, and Python. Typically, the handlers are very simple and only flag a semantic conflict. So, they did not implement specific resolution strategies but serves just to count situations in which they can be applied, which is sufficient for the quantitative analysis. The tool also uses Linux merge tool internally as unstructured counterpart. The tool takes as input the three revisions (base, left and right) that compose a merge scenario, obtained in the previous step, and applies both unstructured and semistructured merge. Finally, we use R scripts to account the results.

IV. Evaluation Results

After performing the study described in the previous section, we present the achieved results before discussing their implications in Section V. Detailed information on each project merge scenario is available on the project's web site (http://goo.gl/hCrI3H).

In the first row of Table I, we summarize the number of merge scenarios according to the results presented by both semistructured and unstructured merge. More specifically, we show for how many merge scenarios semistructured merge reported less, the same number and more textual conflicts than unstructured merge. Besides that, in the second and third rows, we show similar comparisons for conflicting lines of code and conflicting files. For example, 1804 in the first row and column indicates the number of code integrations in which the sum of conflicts headers resulting from semistructured merge is less than that from the unstructured merge. In a similar way, 581 in the second row and column indicates the number of code integrations in which the sum of lines of code surrounded by the conflict headers resulting from both approaches is identical. Finally, 9 in the third row and column indicates the number of code integrations in which the sum of files with at least one conflict header resulting from the unstructured merge is less than that from semistructured merge.

TABLE I. Number of Merge Scenarios where Semistrucutured Merge Reported Less, the Same Number, and More Textual Conflicts, Conflicting Lines of Code and Conflicting Files than Unstructured Merge

	Semistructured Reported Less	**Semistructured And Unstructured Reported The Same Number**	**Unstructured Reported Less**
Textual Conflicts	1804 (55.24%)	1179 (36.1%)	283 (8.66%)
Conflicting LOC	2323 (71.13%)	581 (17.79%)	362 (11.08%)
Conflicting Files	1566 (47.95%)	1691 (51.77%)	9 (0.28%)
Total Merge Scenarios			**3266**

Concerning textual conflicts, semistructured merge reduced the numbers in 55.24% of the sample merge scenarios. In these scenarios, we observed a reduction of, on average, 62.3%, with a maximum of 100% in a merge scenario from *sympy*, minimum of 2.7% in a merge scenario from *OG-Platform* and standard deviation of 23.57%. On the other hand, in 8.66% of the sample merge scenarios, unstructured merge reduced the number of textual conflicts compared to semistructured merge. In these scenarios, we observed a reduction of, on average, 40.49%, with a maximum of 100% in a merge scenario from *nova*, minimum of 0.99% in a merge scenario from *jedis* and standard deviation of 20.09%. In the remainder 36.1% of the sample merge scenarios, both approaches reported a similar number of textual conflicts.

In terms of conflicting lines of code, semistructured merge reduced the numbers in 71.13% of the sample merge scenarios. In these scenarios, we observed a reduction of, on average, 81.04%, with a maximum of 100% in the same merge scenario from *sympy*, minimum of 0.24% in another merge scenario from *sympy* and standard deviation of 13.52%. On the other hand, in 11.08% of the sample merge scenarios, unstructured merge reduced the number of conflicting lines of code compared to semistructured merge. In these scenarios, we observed a reduction of, on average, 43.62%, with a maximum of 100% in the same merge scenario from *nova*, minimum of 0.57% in a merge scenario from *cassandra* and standard deviation of 21.12%. In the remainder 17.79% of the sample merge scenarios, both approaches reported a similar number of conflicting lines of code.

Regarding conflicting files, semistructured merge reduced the numbers in 47.95% of the sample merge scenarios. In these scenarios, we observed a reduction of, on average, 65.51%, with a maximum of 100% in the same merge scenario from *sympy*, minimum of 2.33% in a merge scenario from *OpenRefine* and standard deviation of 25.12%. On the other hand, only in 0.28% of the sample merge scenarios, unstructured merge reduced the number of conflicting files compared to semistructured merge. In these scenarios, we observed a reduction of, on average, 74%, with a maximum of 100% in the same merge scenario from *nova*, minimum of 25% in another merge scenario from *cassandra* and standard deviation of 27.92%. In the remainder 51.77% of the sample merge scenarios, both approaches reported a similar number of conflicting files.

Besides that, seeing that we filtered the merge scenarios by the conflicting ones, we are able to provide evidence about the occurrence of conflicting merge scenarios as it was done in previous studies [2, 5]. That is, from the total of 69924 Merge Commits and 4678 Conflicting Merge Commits observed in our sample, we can determine that approximately 7% of the merge commits lead to conflicts. Similarly, Brun et al. [2] found that conflicts occurred, on average, in 16% of the merge scenarios, and Kasi and Sarma [5] found that from 7% to 16% of the merge scenarios had conflicts. All these numbers rely on the use of an unstructured merge tool (in our case, we use Git built-in merge tool). However, we found that semistructured merge reported occurrence of conflicts in 2362 of the analysed merge scenarios, which represents approximately 3% of the identified merge scenarios. In other words, if the semistructured approach had already been used in the studied projects, the occurrence of conflicting code integrations would have decayed from the previous 7% to remarkably 3% — an improvement of more than 50%. Note that due to the discard of merge scenarios because of the semistructured merge tool's compatibility issues, this number can be considered a lower bound.

TABLE II. RESULTS BY PROJECT. (THE ARROWS INDICATE WHETHER SEMISTRUCTURED MERGE DECREASED, KEPT OR INCREASED THE NUMBERS.)

Project	Unstructured			Semistructured			
	Textual Conf.	Conf. Lines	Conf. Files	Textual Conf.	Conf. Lines	Conf. Files	Sem. Conf.
astropy	28	901	5	13 ↓	378 ↓	3 ↓	2
atmosphere	96	3973	35	66 ↓	1275 ↓	26 ↓	7
Bukkit	27	1464	26	27 ↔	321 ↓	12 ↓	1
cassandra	2286	76816	1258	1576 ↓	31769 ↓	931 ↓	74
clojure	4	39	3	4 ↔	39 ↔	3 ↔	0
cloudify	150	4407	41	93 ↓	1544 ↓	30 ↓	6
CruiseControl.NET	1	8	1	1 ↔	8 ↔	1 ↔	0
cxf	1	219	1	1 ↔	16 ↓	1 ↔	1
django	10	548	9	1 ↓	22 ↓	1 ↓	4
dropwizard	16	479	10	7 ↓	86 ↓	6 ↓	2
Dynamo	336	56682	215	205 ↓	3339 ↓	84 ↓	78
edx-platform	380	7598	222	234 ↓	2620 ↓	114 ↓	55
EventStore	146	5206	99	75 ↓	861 ↓	35 ↓	2
flask	5	14	5	2 ↓	8 ↓	2 ↓	2
gradle	289	10784	206	159 ↓	1770 ↓	119 ↓	24
graylog2-server	104	3387	69	66 ↓	1022 ↓	38 ↓	9
infinispan	90	1433	46	71 ↓	702 ↓	33 ↓	1
ipython	57	4184	47	48 ↓	646 ↓	26 ↓	19
jedis	300	5613	103	276 ↓	2095 ↓	56 ↓	5
jsoup	5	76	3	1 ↓	9 ↓	1 ↓	0
junit	124	4062	75	57 ↓	539 ↓	40 ↓	36
kotlin	162	3880	86	88 ↓	1316 ↓	63 ↓	4
lucene-solr	1633	69035	723	1147 ↓	21678 ↓	485 ↓	53
matplotlib	263	7071	142	157 ↓	2300 ↓	81 ↓	34
mct	44	1344	24	33 ↓	522 ↓	19 ↓	1
mockito	76	992	32	6 ↓	72 ↓	4 ↓	0
monodevelop	911	207013	884	1096 ↑	32380 ↓	565 ↓	111
Nancy	22	536	22	17 ↓	175 ↓	16 ↓	0
netty	168	11922	92	110 ↓	3802 ↓	59 ↓	8
nova	1198	38243	750	871 ↓	10229 ↓	479 ↓	26
NRefactory	40	2156	34	10 ↓	129 ↓	10 ↓	6
NServiceBus	126	4164	109	88 ↓	1015 ↓	71 ↓	13
nupic	49	13896	38	60 ↑	2230 ↓	25 ↓	3
OG-Platform	3530	213733	2155	3266 ↓	51406 ↓	1316 ↓	406
OpenRefine	59	12408	54	106 ↑	3235 ↓	51 ↓	5
opensimulator	5	64	4	3 ↓	72 ↓	3 ↓	0
orientdb	340	15902	189	316 ↓	3948 ↓	111 ↓	68
pandas	28	380	14	18 ↓	348 ↓	6 ↓	5
Questor	78	14743	52	72 ↓	11167 ↓	49 ↓	3
ReactiveUI	34	281	32	2 ↓	27 ↓	2 ↓	0
realm-java	297	12449	190	301 ↑	4305 ↓	145 ↓	21
Rebus	12	341	12	3 ↓	31 ↓	3 ↓	0
requests	16	410	9	12 ↓	131 ↓	5 ↓	1
retrofit	35	671	17	13 ↓	216 ↓	7 ↓	2
roboguice	100	6277	63	105 ↑	1381 ↓	45 ↓	36
Rock	305	13448	170	128 ↓	2332 ↓	67 ↓	24
RxJava	49	2845	34	10 ↓	116 ↓	8 ↓	1
scrapy	5	20	5	0 ↓	0 ↓	0 ↓	11
SharpDevelop	2092	316419	1956	1701 ↓	49216 ↓	839 ↓	304
SignalR	2	14	2	1 ↓	8 ↓	1 ↓	0
slapos	90	5583	53	51 ↓	677 ↓	29 ↓	7
SparkleShare	31	640	11	30 ↓	258 ↓	10 ↓	0
sympy	785	39183	202	836 ↑	17899 ↓	109 ↓	8
taiga-back	9	257	3	1 ↓	5 ↓	1 ↓	1
testrunner	19	1465	10	15 ↓	315 ↓	7 ↓	0
tornado	16	121	16	2 ↓	16 ↓	2 ↓	16
Umbraco-CMS	557	20119	329	398 ↓	6924 ↓	212 ↓	23
voldemort	366	31424	142	257 ↓	4730 ↓	110 ↓	9
WowPacketParser	10	591	5	5 ↓	44 ↓	2 ↓	1
zamboni	4	38	4	2 ↓	4 ↓	2 ↓	0
Total	18021	1257971	11148	14320 ↓	283728 ↓	6581 ↓	1539

In Table II, we further detail what happened in the merge scenarios by showing results by projects. For instance, projects such as *monodevelop* and *nupic* are cases where semistructured merge detects more textual conflicts than unstructured merge. In the next section, we explain that refactoring such as renaming is the reason for this increased number. On the other hand, projects such as *Bukkit* and *Clojure* are cases where semistructured performed similar to unstructured merge. In these cases, the explanation relies on the fact that semistructured merge calls the unstructured mechanism when the order of the statements matters. In addition, from the total columns of Textual Conflicts, we observed that semistructured merge reported approximately 14.3K textual conflicts compared to approximately 18K from unstructured merge, which means that at least 3.7K textual conflicts are ordering conflicts. Likewise, we observe a substantial reduction by semistructured merge in the number of conflicting lines of code (1.25MLOC vs 283KLOC). Indeed, even in some projects where the semistructured approach reports at least the same number of textual conflicts, the number of conflicting lines of code is less than that from unstructured merge. It happens due to the semistructured merge structure-driven and fine-grained nature in which the conflicts respect boundaries of classes, methods, and other structural elements. Another substantial reduction is observed in the number of conflicting files (11148 vs 6581) too. The reduction on the numbers of textual conflicts, conflicting lines of code and conflicting files might indicate a decrease of effort in merging different tasks as we will discuss in the next section. Finally, similar as observed in the original study, the number of semantic conflicts found here is rather low compared to the numbers of ordering conflicts (1.5K versus 3.7K), especially when considering the quite high number of 54 elements that the semistructured merge tool handle with special conflict handlers.

In addition, discerning the significance of data by looking only at their values is not appropriate to extract the important characteristics of a dataset. For this reason, here we manage this shortcoming observed in the original study, and to better interpret data, we carried out a descriptive analysis to observe data tendency and distribution. The three box-plots in Figure 4 indicate that semistructured merge tends to have fewer textual conflicts, conflicting lines of code and conflicting files. The long upper whisker in the box-plots means that both approaches varied amongst the most positive quartile group, and very similar in the least positive quartile group. Note, for example, in the box-plots showing the distribution of textual conflicts and conflicting files according to the merge approach, that, despite the medians being close, the upper limit of the amount of textual conflicts with the semistructured approach is close to the 3rd quartile of textual conflicts using unstructured merge. Besides that, regarding the number of conflicting lines, the overall results from semistructured merge are less than the median of the unstructured one, indicating a more significant reduction.

As a final remark, although semistructured merge has reduced the numbers, such reduction may have been insignificant if compared to the numbers presented by the unstructured mechanism. So, we need to know what is the probability of that relationship of reduction between the merge approaches, described in the achieved results, being due to random chance, and whether there is a good chance that we are right in finding that this relationship exists. Therefore, as each subject of our

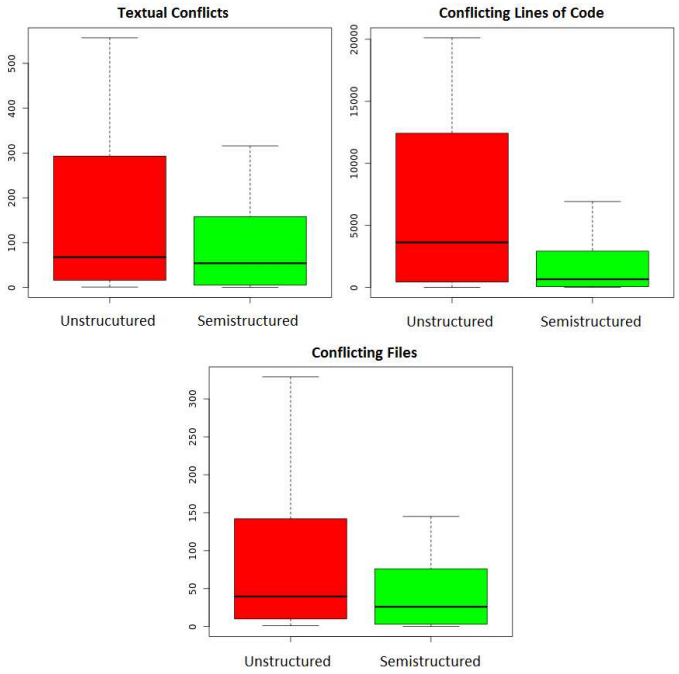

Fig. 4. Box-plots of Sample Projects. (We have hidden outliers for a better visualization.)

sample has two measurements (one with semistructured merge and other with unstructured merge) for the three metrics and deviate from normality, we performed a Wilcoxon Signed-Rank Test on the three metrics from both approaches and obtained a *p-value* of 2.414e-07, 4.21e-11 and 5.245e-11 regarding the number of textual conflicts, conflicting lines of code and conflicting files, respectively. Since these values are lower than 0.05, we cannot accept the null hypothesis of equality of the averages, and therefore there is evidence that semistructured merge has made a significant reduction in the numbers compared to unstructured merge. The original study did not describe hypothesis test, so we cannot compare our results to theirs.

V. Discussion

In this section, we analyse the results presented in the previous section, comparing with those from the original study, and we provide additional discussion on their implications. Besides, we manually analysed selected samples of merged code in order to learn about the influence of the merge approach on the resulting code structure and to better understand our results.

A. *Semistructured merge plays to its sstrengths*

In terms of merge scenarios, the number of merge scenarios in which semistructured merge reduced the metrics was lower in our study than in the original one. Apel et al. [1] observed a reduction in the number of textual conflicts, conflicting lines of code and conflicting files in, respectively, 60%, 82% and 72% of their sample merge scenarios (compared to, respectively, 55.24%, 71.13% and 47.95% in our study). However, apart from our sample being substantially bigger than that from the original study (18x regarding the number of merge scenarios and 2.5x regarding the number of projects), which might explain that variance, the reductions observed in these scenarios

were much more significant in our study. While they observed a reduction of 34%, 61% and 28% (with standard deviations of 21%, 22% and 12%) in the number of textual conflicts, conflicting lines of code and conflicting files, respectively, we observed a reduction of 62.3%, 81.04% and 65.51% (with standard deviations of 23.57%, 13.52% and 25.12%) in our replication. Also notable is the fact that, in the original study, unstructured merge performed better in that it showed less textual conflicts in 28% of their sample merge scenarios, compared to only 8.66% in ours (unfortunately, we do not have numbers regarding the other metrics to compare with). At least in terms of textual conflicts, it means that semistructured merge played to its strengths much more in this study than in the original study.

The observed numbers further confirm the original hypothesis that many of the conflicts that occur in merging revisions are ordering conflicts — more specifically, at least 21% (or 3.7K) of the reported textual conflicts— which can be resolved automatically with semistructured merge. We also found that an additional fraction of conflicts can be potentially resolved with language-specific conflict handlers. These findings reinforce the benefits of exploiting the syntactic structure of the artefacts involved in a code integration.

With respect to the positive implications of the results, a previous study [18] suggests that the integration effort is the number of extra actions (additions, deletions or modifications on the artefacts) that the developer had to do during the integration to conciliate the changes made in revisions developed concurrently (here, the left and right revisions of a merge scenario). A more recent study [19] correlates the number of conflicts to that metric. Therefore, reducing conflicts might indicate effort reduction. By following this reasoning, since semistructured merge reduced the number of conflicts compared to the unstructured approach, it would be also reducing the integration effort. Moreover, the substantial reduction made in the conflicting lines of code could support this reasoning if we consider that the greater is the number of conflicting lines of code that the developer has to deal with, the greater is the developers' effort to integrate the code contributions because conflicting lines of code amounts to the size of the conflicts. The shortcoming of this metric is that it means only part of the time that the developer takes editing the code, it does not consider the time that the developer took reasoning about those activities, for instance. This way, this editing time can mean just one part of the total integration effort.

Besides that, although we are comparing the approaches regarding the number of textual conflicts, conflicting lines of code and conflicting files, these metrics do not provide guarantees with respect to the occurrence of false positives and false negatives. To that end, we would have to manually analyse conflicts from a small sample, perhaps with the help of specialists, to possibly establish what is indeed false positives and false negatives, which would divert the focus of the replication. However, it is still possible to reason about it. That is, some of the reported conflicts are false positives (e.g., the ordering conflicts): they do not reflect interference between development tasks. Thus, resolving them is an unnecessary effort. Since we found that semistructured merge reduces such conflicts, there is an indicative that semistructured merge decreases unnecessary integration effort. On the other hand, there are conflicts that are not detected and, in turn, reflect interference between developers' tasks (false negatives). For example, when two developers add methods with the same signature in different parts of the code. These cases might be harder to track, understand and resolve, leading to build and behavioral errors. In brief, false negatives compromise quality and correctness of the merging process. Regarding that, as mentioned before, when semistructured merge is not able to merge a software element, such as method bodies with statements, it represents the elements as plain text and uses the conventional unstructured merge instead. That is, when the conflict is not an ordering conflict neither is there a conflict handler available, semistructured merge works the same as the unstructured one. This way, we believe that semistructured merge does not worsen correctness compared to the unstructured approach.

Finally, in the same way as the original study, we could observe that semistructured merge, due to its structure-driven and fine-grained nature, leads always to conflicts that respect boundaries of classes, methods, and other structural elements. This is not the case for unstructured merge, the conflicts are typically larger and often crosscut the syntactic program structure, which makes them harder to track and understand. It happens because the unstructured approach resolves conflicts via textual similarity, based on the position of the changes in the text, without any knowledge of the underlying language. Thus, it is common to find conflicts involving mismatched syntactic structures, such as a conflict involving the signature of a method and a loop statement simply because they were in the same region of the code, which hardly makes sense. Respecting structural boundaries (i.e., aligning the merge with the program structure) might be beneficial, because, this way, developers could understand conflicts in terms of the underlying structure.

B. Semistructured keeps or increases the number of conflicts

In cases where the approaches showed a similar number of textual conflicts, it happened because those conflicts occurred inside method bodies. In this situations, the statements' order matters, and thus semistructured merge calls the unstructured merge. We could see that in revisions from *clojure*, *cassandra*, *flask* and so on.

Finally, in cases where semistructured merge increases the number of textual conflicts or conflicting lines of code, Apel et al. [1] found that refactorings, such as *renaming*, challenge semistructured merge due to the use of superimposition to merge revisions. If a program element is renamed in one revision, the merge algorithm is not aware of this fact and cannot map the renamed element to its previous version. This results in a situation in which there is, in one revision, an empty or non-existent element. Besides, if the other revision changes the original method body, when semistructured merge tries to integrate the methods, it will notify a conflict because the three versions become different (the original, the body-changed and the renamed). In Figure 5, we illustrate this situation using a snippet of code taken from a merge scenario of *NServicebus*. One of the developers (left) changed the signature of the *Init* method, while another developer (right) kept the old signature, but added an extra assignment statement. Since the changes were made in different lines of the text, unstructured merge did not report any conflict, however semistructured merge did report a conflict due to the modification in the method's signature in the left revision and the modification in the

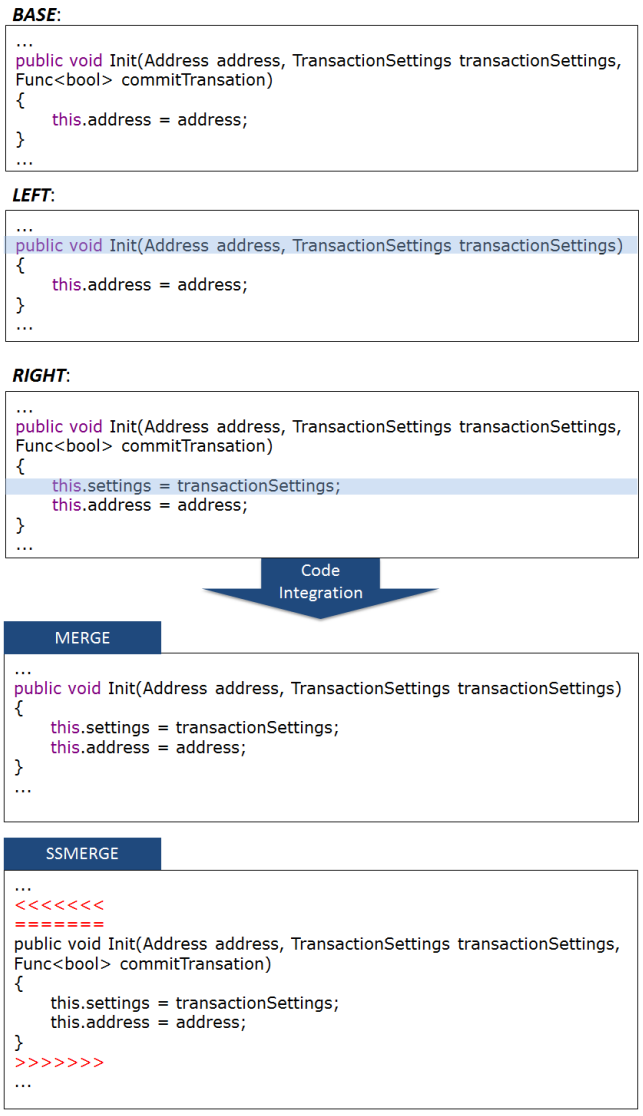

Fig. 5. Renaming Issue Example taken from NServiceBus project.

method's body in the right revision. It is important to notice that, in Java, a method is identified by its name and the types of the formal parameters. If one of the two differ between two declarations, semistructured merge current implementation cannot match them anymore.

The issue gets worse when a directory is renamed — instead of a method's signature. This happened, for instance, in merge scenarios from *monodevelop*, *kotlin* and *graylog2-server*. In these cases, unstructured merge reports a large conflict for each file into the directory because it cannot map the files of the renamed directory to the corresponding files of the other revision and uses empty files instead. The same happens in semistructured merge, except that the conflicts are not reported per file but per method or constructor in the file. This results in more conflicts but the overall number of conflicting lines is smaller than in unstructured merge. The reason is that unstructured merge has a file level granularity, while semistructured merge has a structural element level granularity. Therefore, in these cases, unstructured merge flags entire files as conflicts, and semistructured merge flags only individual structural elements such as methods. A more recent study [20] presents an approach for detecting differences, moves, and refactoring-related changes on source code through dynamic programming algorithms to find the *longest common subsequences* [21] between the files; one can improve semistructured merge tool with a similar approach.

Whereas in the original study renaming is seen as an issue, that is, a false positive, which should not happen, if we consider the definition of *interference* given by Goguen et al. [22], where a conflict exists if the work of a developer interferes in the work of another, a renaming conflict can be seen as true positive. Figure 5 also illustrates this example: the left developer might not be waiting for the extra assignment statement, and the right developer probably would call the *Init* method with three parameters, as in the original version. This is an interference captured by semistructured merge but not by the unstructured one. In this case, the unstructured merge output of this example also serves to illustrate a false negative.

VI. Threats to Validity

Since our study is a replication of the research made by Apel et al.[1] it is natural that our study suffers from some of the same threats to validity. This holds particularly to the threats to construct and external validity. However, in our replication, we were able to improve internal and external validity.

Construct Validity: Apel et al.[1] remarks that the output of semistructured merge in the presence of renaming is not satisfactory, but it still allows us to detect conflicts properly and to incorporate them in our data. A threat to the construct validity is that the number of conflicting lines of code may be estimated too low, because the renamed element is not considered. However, after code inspection, we believe that the occurrence of renaming-related conflicts would has little potential to impact the reduction of approximately 1MLOC we found here regarding the number of conflicting lines of code if the renamed version was considered.

Internal Validity: a potential threat to internal (and external) validity is also our approach to select conflicting merge scenarios. The problem is that some distributed development models make use of mechanisms, such as `git rebase`, that can rewrite the repository history, making impossible to identify merge commits of the old history. Otherwise, we could have more merge scenarios and more conflicts to analyse. This way, we have explored a of what really happened in projects' history. The impact of an increased sample on the results presented here is hard to predict. However, we still believe that we performed better than the original study in this aspect because we only analysed real merge scenarios from the sample projects instead of using "speculative" merge scenarios as happened in the original study.

Besides, one can argue that we bias the results for the reason that we discard merge scenarios not supported by the semistructured implementation used in the study. Nevertheless, we analysed source code carefully to understand this problem and we found that most of the unsupported content is language's constructions that happen inside method bodies, where semistructured merge works exactly like the unstructured one. Thus, it means that in such cases the results presented by both approaches would be almost the same.

External Validity: to increase external validity, we collected a substantial number of projects and merge scenarios

written in different languages and of different domains, with a number 2.5 times bigger regarding to the number of projects and 18 times bigger regarding the number of merge scenarios compared to the original study.

VII. Concluding Remarks

During the integration of code contributions resulting from developers' tasks, one likely has to deal with conflicting changes and dedicate substantial effort to resolve them. To reduce such effort, while unstructured merge tools try to automatically solve part of the conflicts via textual similarity, structured and semistructured merge tools try to go further by exploiting the syntactic structure of the artefacts involved. In this paper, with a replicated experiment of Apel et al. study [1], we evaluate the semistructured approach in a sample 2.5 times bigger than the original study regarding the number of projects and 18 times bigger regarding the number of merge scenarios. Subsequently, in proportions far greater than the original study, we observed a significant decrease in the number of textual conflicts, conflicting lines of code and conflicting files by using semistructured merge compared to the unstructured merge. We also observed that an additional fraction of conflicts can be potentially resolved with language-specific conflict handlers. Additionally, our study suggests that the use of semistructured merge might decrease the developers' integration effort not compromising software correctness. Finally, we state that if semistructured merge was used in place of the unstructured one, the occurrence of conflicting merge scenarios would drop by half. As a future work, we plan to conduct a more detailed qualitative study regarding the conflicts identified in this paper to find out which conflicts appear more frequently in practice, which are false positives, how to avoid them during the collaborative development; and which kind of interferences the approaches are not able to detect (the false negatives) and how to improve them to do so.

Acknowledgment

We would like to thank FACEPE Brazilian research funding agency grants IBPG-0777-1.03/13, INES, funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08, for partially supporting this work. Also, we thank Sven Apel for the personal support concerning the original study, and Thais Burity and Klissiomara Dias and all anonymous reviewers for the comments that certainly helped to improve the paper.

References

[1] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, "Semistructured merge: Rethinking merge in revision control systems," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. ACM, 2011, pp. 190–200.

[2] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. ACM, 2011, pp. 168–178.

[3] B. K. Kasi and A. Sarma, "Cassandra: Proactive conflict minimization through optimized task scheduling," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, pp. 732–741.

[4] T. Zimmermann, "Mining workspace updates in cvs," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, ser. MSR '07. IEEE Computer Society, 2007, pp. 11–.

[5] A. Sarma, D. Redmiles, and A. van der Hoek, "Palantir: Early detection of development conflicts arising from parallel code changes," *IEEE Trans. Softw. Eng.*, vol. 38, no. 4, pp. 889–908, Jul. 2012.

[6] S. Apel, O. Lessenich, and C. Lengauer, "Structured merge with auto-tuning: Balancing precision and performance," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. ACM, 2012, pp. 120–129.

[7] J. E. Grass, "Cdiff: A syntax directed differencer for c++ programs," in *C++ Conference*, 1992, pp. 181–194.

[8] B. Westfechtel, "Structure-oriented merging of revisions of software documents," in *Proceedings of the 3rd International Workshop on Software Configuration Management*, ser. SCM '91. ACM, 1991, pp. 68–79.

[9] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Jdiff: A differencing technique and tool for object-oriented programs," *Automated Software Engg.*, vol. 14, no. 1, pp. 3–36, Mar. 2007.

[10] T. Mens, "A state-of-the-art survey on software merging," *IEEE Trans. Softw. Eng.*, vol. 28, no. 5, pp. 449–462, May 2002.

[11] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, ser. MSR '09. IEEE Computer Society, 2009, pp. 1–10.

[12] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, "How do centralized and distributed version control systems impact software changes?" in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. ACM, 2014, pp. 322–333.

[13] B. O'Sullivan, "Making sense of revision-control systems," *Commun. ACM*, vol. 52, no. 9, pp. 56–62, Sep. 2009.

[14] S. Khanna, K. Kunal, and B. C. Pierce, "A formal investigation of diff3," in *Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science*, ser. FSTTCS'07. Springer-Verlag, 2007, pp. 485–496.

[15] J. Buffenbarger, "Syntactic software merging," in *Selected Papers from the ICSE SCM-4 and SCM-5 Workshops, on Software Configuration Management*. Springer-Verlag, 1995, pp. 153–172.

[16] S. Apel and C. Lengauer, "Superimposition: A language-independent approach to software composition," in *Proceedings of the 7th International Conference on Software Composition*, ser. SC'08. Springer-Verlag, 2008, pp. 20–35.

[17] M. A. Rodriguez, "Graph databases: Trends in the web of data," *KRDB Trends in the Web of Data School-Brixen/Bressanone, Italy*, 2010.

[18] J. a. G. Prudêncio, L. Murta, C. Werner, and R. Cepêda, "To lock, or not to lock: That is the question," *J. Syst. Softw.*, vol. 85, no. 2, pp. 277–289, Feb. 2012.

[19] R. d. S. Santos and L. G. P. Murta, "Evaluating the branch merging effort in version control systems," in *Proceedings of the 2012 26th Brazilian Symposium on Software Engineering*, ser. SBES '12. IEEE Computer Society, 2012, pp. 151–160.

[20] F. F. Silva, E. Borel, E. Lopes, and L. G. P. Murta, "Towards a difference detection algorithm aware of refactoring-related changes," in *Proceedings of the 2014 Ninth International Conference on Availability, Reliability and Security*, ser. ARES '14. IEEE Computer Society, 2014, pp. 111–120.

[21] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Commun. ACM*, vol. 20, no. 5, pp. 350–353, May 1977.

[22] J. A. Goguen and J. Meseguer, "Security policies and security models," in *2012 IEEE Symposium on Security and Privacy*.