To Branch or Not to Branch?

Rahul Premraj
VU University Amsterdam
rpremraj@cs.vu.nl

Antony Tang
VU University Amsterdam
atang@cs.vu.nl

Nico Linssen
Océ Technologies
nico.linssen@oce.com

Hub Geraats
Océ Technologies
hub.geraats@oce.com

Hans van Vliet
VU University Amsterdam
hans@cs.vu.nl

ABSTRACT

The study of Software Configuration Management (SCM) has so far focused on the use of tools, SCM models, strategies, patterns or best practices. There are few industry studies on how an organization actually practices file branching and merging. In this empirical study at Océ, we have observed that some developers create branches freely without regards to their consequences on merging. This contradicts recommended best practices and SCM processes. So we investigate if there are hidden costs in propagating or merging changes at Océ. The investigation led us to understand that branching and merging can be done freely under certain circumstances to provide development concurrency. However, some files cannot be branched freely and it may be better to use recommended practices to edit them. Some roles were also noted to be more affected by branching of files.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Software configuration management

General Terms

Experimentation, Management

Keywords

Software Configuration Management Practice, Branching, Merging, Agile Development

1. INTRODUCTION

Software configuration management (SCM) systems are designed to facilitate the management, coordination, sharing, and changing of software artifacts by different people including developers, architects, build managers, testers, and others. In the past, SCM has been studied in terms of its tools [8], practices [2,15], and processes [7] and several modern applications are currently in wide use such as CVS, SVN, Git, Bazaar, Mercurial, and Synergy.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSSP'11, May 21–22, 2011, Waikiki, Honolulu, HI, USA
Copyright 2011 ACM 978-1-4503-0730-7/11/05 ...$10.00

Limited effort, however, has been invested into studying how people actually use SCM in practice, especially in environments where parallel development, i.e., concurrent editing of the same file by more than one developer, is inevitable due to the size of system and teams, and consequently software artifacts are shared by many developers [1]. The ways developers coordinate amongst themselves to co-edit files may have implications on the cost and correctness of merging. For instance, one developer checks out a file for a long time, another developer who wants to modify the same file must *branch* in order to carry out his work. This could either increase development overhead [9], or it may help co-development concurrency [11,13].

Certain SCMs (e.g., IBM Synergy) allow parallel development by implicitly and automatically creating a copy of the software artifact(s) when it is shared by more than one developer. Creation of the copy is referred to as *branching* and can occur without the knowledge of the second developer that someone else is editing the file in parallel. Other SCMs such as CVS and SVN do not create a branch in such situations, but raise conflicts when changes are made to the same parts of the file. In both types of SCMs, the two copies need to be merged so that the changes made in one copy are propagated to the other copy (referred to as *merging*) [5]. Such practice is referred to as file-oriented branching since the granularity of branches created is at the file level [1]. Note that our study strictly investigates the effects of file-oriented branching as opposed to project-oriented branching where an entire project is branched, for instance, to maintain a specific release [4]. The manner to facilitate parallel development differs between SCMs, but the workflow is similar — concurrent editing of files and integration of changes.

In an industry case, we have observed a lot of branching and merging activities. All files together in this study have over 1,790 branches over a period of 18 months. The top 10 most heavily branched files have an average of 39 branches over this period. In some cases, more than one parallel branch exists. With so many branches created, we want to know what this would mean for software development coordination and cost. The way developers in this organisation branch their files appears to contradict industry best practices [3, 10, 15], but at the same time their current practice does not seem to have impeded their development. This contradiction motivated us to study their SCM practices. In this study, we try to understand these questions:

- We want to understand why branching occurs. What kind of files are branched and under which circumstances are they branched? Also we wish to investi-

gate what is the cost of merging and which costs can potentially be saved?

- Agile development provides an environment where developers tend to co-edit files because of the way development sprinting works. If file co-editing is inevitable, what would be a good practice(s) to balance merging cost with the convenience of branching?

In order to understand these issues, we have conducted both quantitative analysis and interviews. The quantitative analysis was used to investigate issues such as who does the branching and how much time does one spend on merging, and are there differences in the way different files are treated? We conducted interviews to understand the roles and tasks of the developers so that we can identify issues of branching and merging. In this study, we have learned that certain types of files can be branched and merged freely under specific circumstances, whereas some types of files had better follow SCM best practices to control merging cost.

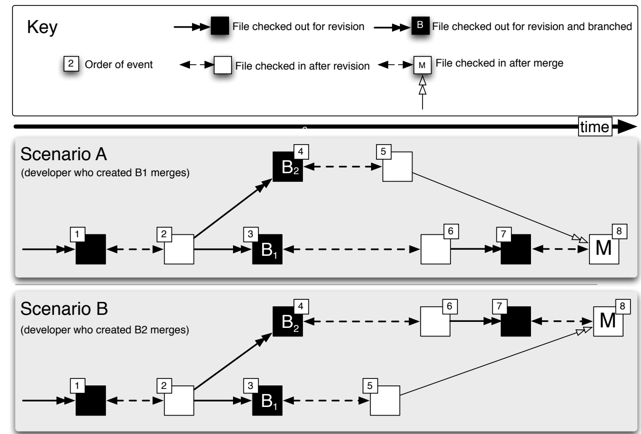

Figure 1: Typical Branching and Merging Scenarios

2. STUDY ENVIRONMENT

Océ Technologies (www.oce.nl) produces high-end printers to serve the business markets for high-volume printing, wide-format printing and office printing. Printer software is one of the many components in a printer. The software is responsible for accepting requests, controlling print jobs, rendering images and controlling devices such as the print engine and finishers. The software system we studied is a mature product comprising over 40,000 program source and other files.

Agile development environment. The software programs and files are developed, maintained, and shared by over 100 developers, integrators and testers spread over three development sites — we refer to these sites as Sites A, B, and C. The company uses an agile development method. A release is typically 6-8 weeks long and each sprint is typically 2 weeks long. Scrum meetings are conducted daily within a team and regular scrum of scrum meetings are conducted between teams, both within and across sites, to discuss relevant issues. The agile method and the development culture at Océ endow ample freedom to their developers to plan their activities. Development teams and individual developers would be assigned tasks and they have certain freedom to schedule their tasks. Since development scheduling is largely autonomous, the co-editing of shared files is largely unplanned. This environment is therefore conducive to branching.

Development workflow. The typical workflow of developers at Océ is as follows: (a) by agreement, a number of tasks are allocated to each developer; (b) a developer would schedule her own activities to achieve all the tasks at the end of each sprint; (c) a developer would decide what files are required to be edited and added, then s/he would *check-out* the latest revision of file(s) from the SCM (IBM Rational Synergy) system for modification; (d) depending on the type of file, different tools can be used to facilitate check-out. For instance, a developer may use VisualStudio to check-out a source file from Synergy. If the file is already checked-out, the developer would be asked by the system whether a branch can be created; (e) after a file has been modified, the tool can auto-merge changes if concurrent editing on the same file by another person has occurred. Should a user desire to merge a file manually because of conflicts in the editing, this is also supported. Typically the last person who checks in a file would do the merging.

If we want to guarantee that there are no editing conflicts, a file should be checked-out by only one developer at a time and other developers must wait until this developer has finished editing. This is not always feasible. If another developer checks-out an already checked-out file, *a branch is created*. Both developers eventually complete their changes and check-in their respective file revisions. But the last of the two developers to check-in the changes is prompted of an existing parallel revision that must be merged (akin to a conflict and resolve situation in version control systems such as CVS and SVN). This developer then checks-out a new file revision (the merge) to consolidate the changes made by the other developer and checks-in the merged revision back into the SCM.

File co-editing can happen in many scenarios. For instance, two developers can be editing the same program or configuration files because of overlaps in their development work. A developer may be developing new software whilst another developer may be fixing a bug. An integrator may be editing a configuration file for integration testing whilst a developer is still doing development. Furthermore, a designer or software architect may be refactoring and merging files in order to incorporate new changes. These different scenarios may also have some bearing on the way files are branched and merged.

In Figure 1, we illustrate two common scenarios of branching and merging. In scenario A, a file is checked-out, modified, and checked back in the SCM (blocks 1 and 2). Thereafter, two developers (Developers 1 and 2) check-out the file (blocks 3 and 4) together and thus create parallel versions. Note that the branch created in block 4 is slightly to the right of the branch in block 3 indicating that it is Developer 2 who checked-out the file second and created the branch (see timeline between the key and scenario A). Next, Developer 2 completes her changes and checks them in the SCM in block 5 before Developer 1 checks in her changes in block 6. Upon checking-in, Developer 1 is notified that a branch was created and must be merged — she then proceeds merging her changes with those from Developer 2 in blocks 7 and 8. So in this scenario, Developer 1 who was not responsible for creating the branch ends up having to merge the changes.

In scenario B, Developer 2 is the developer who creates the branch and she has to merge the changes.

3. SCM OF THE PRINT SYSTEM

We study the Océ print system and analyze the data from their SCM that contained development activity (changes made to files) from three active projects from January 2009 to June 2010 . The data included the following information:

Filename: name of the file added or changed,

Revision: revision number of the file,

Component: component to which the file belongs to,

Owner: name of the developer who added/changed the file,

Site: the site location (sites A, B, or C) where the addition/change was made,

Timestamps: check-out and check-in timestamps for the file revision,

Successor: the next revision number for a file's revision.

In the above data, file versions that are branches and merges are not explicitly flagged; however, they could be reliably inferred from the last information item (Successor) in the above list. A file revision is typically succeeded by another single revision. But when a developer checks-out an already checked-out file, a file revision is succeeded by two (or more) revisions. We refer to such a file revision as a *branch point*; the succeeding revisions are the branches. Likewise, merges can be identified as file revisions that have more than one preceding revision.

The data comprised 36,327 unique files that were added or modified since January 2009 with a total of 73,373 revisions. The 96 developers at site A created 77% of the file revisions, the 43 developers at site B created 17%, and the 28 developers at C created 6%.

Issues noted in the data.

On inferring branches in the data using the method described above, we found that 370 files were branched at least once and 1,790 revisions created for these files were branches. While the percentage of branched files and revisions is low, several observations about the pattern of branching and merging raises questions that are worthy of investigation. For instance, we observed that:

- while some files were branched only once, over half of the revisions for some other files were branch points,
- files of certain types are more prone to branching than others (more on this in Section 5.4),
- only 37% of the 98 components in the project had branches. But the number of branches across these components varied a lot — some components had more than 17% file revisions that are branch points, while other components with comparable number of file revisions had less than 1% that are branch points,
- some developers create more branches, in proportion to their overall development activity, in comparison to other developers (more on this in Section 4).

The above observations suggest that specific factors, individually or coupled, influence when branches are created. While the proportion of branches is low, it can create large overhead for the company — nearly all branches must be merged (1,528 such instances were found in the data), merges could have been avoided if no branches were created in the first place. These merges cost developers time (see Section 5) and thus warrant an investigation to understand the reasons for the above noted observations.

In order to identify the factors and underlying reasons for the creation of branches and estimate related overheads resulting from performing merges, we performed a qualitative analysis by conducting a survey with the developers at Océ. We further conducted a quantitative analysis on the data and compared the results to the responses from the survey and checked the extent to which they match. Any discrepancies may point to potential areas for improvement in the software development process so as to minimize the burden originating from branching and merging files. The remainder of the paper focuses elaborating upon our analysis and summarizing our findings.

4. QUALITATIVE STUDY

We surveyed 16 software personnel in the company to understand the underlying reasons for the above issues found in the data. The focus of the survey was to learn the developers' views about branching and merging files in their work environment and their experience of the development overheads from branching and merging.

4.1 Method

In order to examine developers' behaviour with respect to branching and merging files, we carefully selected some of them to study on the basis of their branching and merging patterns to be found in the data. The selected developers can be classified into six groups based on how frequently they branched and merged files:

Group A. Developers who created most branches.

Group B. Developers who created the highest percentage of branches relative to the total number of files revisions created by them

Group C. Developers who created many file revisions (indicating that they are active developers) but have created no or very few branches.

Group D. Developers who merged most branches.

Group E. Developers who had the highest percentage of merges relative to the total number of files revisions created by them.

Group F. Developers who merged a lot of branches that they had no original involvement in.

From each group, we selected the top four developers to contact for the survey. A person may belong to more than one group. For instance, someone who is a top brancher (Group A) may as well have merged several files (Group C). Hence the final list consisted of 17 developers of which 16 replied. These 16 developers have different roles in the company — these roles include programmers/developers, testers, integrators, and architects.

4.2 Scope of the Interview

The main objectives of the interviews are to consider the following aspects:

Configuration practice. Understand the current process and company mandated practices to evaluate their influences on the branching and merging activities.

Situations. Understand the situations and criteria under which developers would decide to branch a file.

Resulting impact. Understand whether developers know about the impact of branching and merging on the project.

The above six groups of people are designed to include those who frequently branch and merge (except for Group C). From the people in Groups A, B, and C, we wish to find out why (or why not) do they branch files so frequently and whether they follow any strategy when sharing files with other developers. A comparison in the strategies between developers of Groups A and B with those from Group C may reflect on plausible process solutions to decrease the occurrence of branches in the future. From the people in Groups D, E, and F, we wish to understand whether excessive branching has any undesired impacts on their workflow, the system, or the development process as a whole. Their responses would help investigate whether branching and merging files has any consequences or whether it is trivial and safe.

Keeping the scope of the survey in mind and the information we wished to learn from the developers from each group, we designed two sets of questions, one for the branchers and another for the mergers. Developers were briefed on the motivation of the research so they could better align their responses to the information we sought.

The following are the questions asked to branchers (Group A, B and C):

Q1. *What process do you follow when you check out a file that could be shared by other developers?* The purpose of this question is to find out whether they create branches indiscriminately or exert some caution when checking out files.

Q2. *When and why do you create branches?* The purpose of this question is to find the circumstances under which a file is branched: time pressure, nature of file, "couldn't care less" attitude. In case of Group C, we asked them why they do not branch.

Q3. *Do you check-in your files quicker when you know that you are working on a branch?* The purpose is to understand if developers check-in faster so as to avoid having to merge themselves.

Q4. *Are you aware of any best practices related to branching files?* The purpose of this question is to check if they are aware of best practices related to branching and merging. If they are aware of any best practices, we asked them whether they follow best practices or not, and why?

Q5. *What are the effects of branching on your project?* This question is to check if they are aware of any potential negative consequences of branching.

The questions asked to the mergers (Group D, E and F):

Q6. *How do you prepare for a merge? Why?* This question is to understand the merging process in order to understand if there are potential issues that may arise from this process.

Q7. *What technical and coordination issues have you encountered when merging files?* This question is for understanding if there are any negative consequences from excessive branching.

Q8. *Does merging consume a lot of time? If so, how can the situation be improved?* This question is about any improvements that may be helpful if problems exist in the current practice.

4.3 Responses to Branching Related Questions

Ten developers from Groups A, B, and C responded to our survey. In this section, we summarize their responses for each question:

Q1. *What process do you follow when you check out a file that could be shared by other developers?* Nearly all developers replied that they do not perform any additional cautionary steps before checking-out files to work on them. However, some developers exercise more caution when branching: (a) A configuration manager replied that before beginning to work on a file, he checks its recent history to understand past changes, but then proceeds to check out the file even if a branch is created; (b) A software engineer replied that he typically informs his colleagues on the files he is currently working on so as to avoid/minimize branches created by others; (c) two other software engineers claimed that if the SCM warns them that a branch will be created if they check-out the file, they check with other developer to get an estimate on when the changes will be completed. If possible, they wait to avoid creating a branch. In general, creating a branch does not appear to be an issue for the developers.

Q2. *When and why do you create branches?* Most survey respondents answered that they do not hesitate to create branches — one response was *"whenever I want"*. Bug-fixing and new features were the two tasks cited for which branches are most often created. An integrator replied that he simply branches a file if he anticipates his changes are small and quick. Another reason cited for branching was that a file may have been checked-out by someone else for a long period of time and the changes can no longer wait.

As to the latter question on why they create branches, the most common answer was that parallel development increases work efficiency. Developers don't have to wait for others to finish their changes before commencing their own. Another reason cited for branching was a technical issue — sometimes, the SCM databases[1] from different sites may not have synced properly or on time, as a result of which a developer may unawarely create a branch.

Q3. *Do you check-in your files quicker when you know that you are working on a branch?* This question was a little controversial in that responses may partially reflect on the work ethics of the respondents. Nearly all respondents replied that they do not hurriedly check-in files when they have created branches. Some respondents elaborated by adding that the quality of the

[1]For several business reasons, developers at different sites work with separate databases that are routinely synced several times a day.

checked-in code is paramount, irrespective of whether quality comes at the cost of having to do the merge the files themselves.

However, two developers admitted that at times, they check files back into the SCM to save themselves from the complexity of merging. To quote one reply:

"I normally check with the co-developer, but yes, I might sometimes be egoistic by checking in quicker so that the merging has to be done by the other. This however also depends upon the level of trust I have in the co-developer doing correct merges of my changes."

Q4. *Are you aware of any best practices related to branching files?* All developers, but one, were not aware of any best practices on branching (and merging) such as those postulated by Appleton et al. [1]. Their workstyles appeared individualistic and driven by *"commonsense"* experiences and their environment. The one software engineer who claimed to be aware of best practices cited several such as merge early and often, decomposition of work into little tasks, and attaching detailed comments to tasks. It is noteworthy that this software engineer belonged to Group C (the one with none or few branches).

Q5. *What are the effects of branching on your project?* The responses to this question indicated that only a few respondents conceived branching to be a problem, although many admitted that on rare occasions merges can become very complex because of the degree of changes in the branched files, especially when the code is refactored. Other downsides cited in the responses include merges take time (reply from a configuration manager), nightly build failures because of pending merges (the build system cannot compile with parallel file versions), increased risks of introducing regressions into the merged file, increase in maintenance overhead, and the risk of human-error during merging (an architect admitted to one such incident that costed a lot of time to identify and fix the issue). An experienced software engineer replied that branching is not a problem at all because *"branches do not necessarily conflict in locations"*, and thus merges typically take little time. Two respondents replied by advocating the benefits of branching files:

"[Branching] Speeds-up the project [...] instead of waiting for a check-in of a colleague."

"We work in parallel on different tasks on the same file. This way each developer has complete liberty and is independent of others and this optimizes the working process."

The overall behaviour of the respondents with respect to branching can be characterised as "opportunistic": the developers create branches when their individual tasks demand so. Their practices appear to contradict recommended best practices [15] where developers branch only when necessary and according to certain policies. Some developers did point to potential problems that occasionally come up such as complex merging and human error. We also noted from the responses that developers with roles such as configuration managers and integrators, exerted slightly more caution with respect to branching and merging. Having said that, some developers also admitted to be wary of branches.

4.4 Responses to Merging Related Questions

This section summaries the thirteen responses from Groups D, E, and F to our survey:

Q6. *How do you prepare for a merge? Why?* The most common answer received is that they perform the merge independently. Only on rare occasions, when in doubt, do they discuss the details on how to merge with the other concerned developers. Many respondents reiterated the fact that merges are often trivial and an elaborate coordination process is unnecessary. As a means to further ease merging, Synergy (the SCM used at Océ) provides automated tool support for merging files which is a widely used feature (unless the merge is non-trivial).

Q7. *What technical and coordination issues have you encountered when merging files?* No severe issues related to merging were cited in the replies. A couple of respondents mentioned that it may sometimes be a little challenging to arrange meetings with other developers because of holidays and time schedule, but this is a minor issue. Another developer also cited that merging files using different character encodings are not rare in `xml` files, which can cost a little more time to resolve. No other issues were cited.

Q8. *Does merging consume a lot of time? If so, how can the situation be improved?* Most developers view that merging typically costs little time, especially with the use of the automated merge tool. However, under some occasions (as cited in responses to Q5.), merging can be complex and must then be performed manually. Respondents suggested how the efforts of merge can be minimized. The suggestions included clear delineation of responsibilities of developers across files, splitting large (non-program) files such as `xml` files by developers, roles, or tasks, and redesigning program files so as to reflect both functionality and the team structure in the company. We discuss these solutions in more detail in Section 6.

The responses of the branchers and the mergers agree with each other. They are of the opinion that branching and merging is not an issue in the company, except in occasional cases when merging may become complicated. There could be several reasons for this opinion such as implicit ownerships of parts of a file to avoid conflicts, the automated merge tool support widely used, and perhaps a conducive team environment that poses no serious challenges to coordinating merges. In the next section, we verify the data quantitatively to check whether branching and merging poses any problems at all, and if so, under what circumstances.

5. QUANTITATIVE STUDY OF BRANCHING AND MERGING

5.1 Overhead of Branching and Merging

Recall that most developers considered merging as trivial and that it only negligibly increases the overhead in the development cycle (Section 4.4, Q8). We verify this common claim by investigating the data quantitatively and find

out how much overhead was incurred in the duration of 18 months. Overhead, in our context, is the amount of time taken to merge files because this is additional work that could have been avoided if there were no branches in the first place. It could be computed by leveraging the check-out and check-in timestamps available in the data. There can be other types of overheads, e.g., defects introduced in merges, but it was not possible to compute such overheads from the data.

We found a total of 1,528 merges for 648 files in the data (note that all merges related to only two versions of a file). The minimum amount of time spent by a developer to merge a file was 3 seconds. Quite likely, this merge may have been performed using the automated tool support in the SCM that developers mentioned in the survey (Section 4.4, Q6). Also, the changes to be merged may have been made in different parts of the file that made the merge straightforward. However, the median time to merge files in the data is 5 minutes 25 seconds. While this duration does not appear to be substantial, it raises questions about the abilities of the automated merge tools when changes are slightly more complex and some human involvement becomes necessary. The total time spent on merging files up to the median value amounted to 21 hours 20 minutes.

But further investigation into the data paints a different picture of the overhead from merges. The mean value of all merge times is 6 hours and 48 minutes suggesting that the distribution of the duration is highly positively skewed, i.e., the median lies to the left of the mean and that there are several large merge duration values. In fact, the longest duration taken to perform a merge is 16 days; even more, the sum of the durations above the median value is 432 days. This distribution suggests that while some merges were trivial and could be performed automatically with tool support and limited human intervention, several other merges were a lot more time-consuming and substantially contributed to the project's overhead.

The duration of merges may have depended upon the extent of changes made in the branches, i.e., large changes in the branch will require more time to carefully merge them. Recall that refactoring in a branch was cited to be one cause for long merge times in the survey. We compared the duration of the branches (assuming longer durations suggest more severe changes) and respective merge times to verify whether there is a relationship between the two. Figure 2 plots the ranked duration of branches on the x-axis and the ranked duration of the corresponding merges on the y-axis (the values were ranked to normalize the raw data that was originally not normally distributed). The thick line running across the graph is a loess smoother[2] that indicates the general direction of the relation between the data points. One can see from the spread of the data points in the figure that branches with small durations (suggesting smaller changes) generally take less time to merge, while branches with longer durations (suggesting substantial changes) take longer time to merge. This relationship is confirmed by the upward moving loess smoother. The Spearman's correlation between the branch and merge duration times was 0.36 and the Pearson

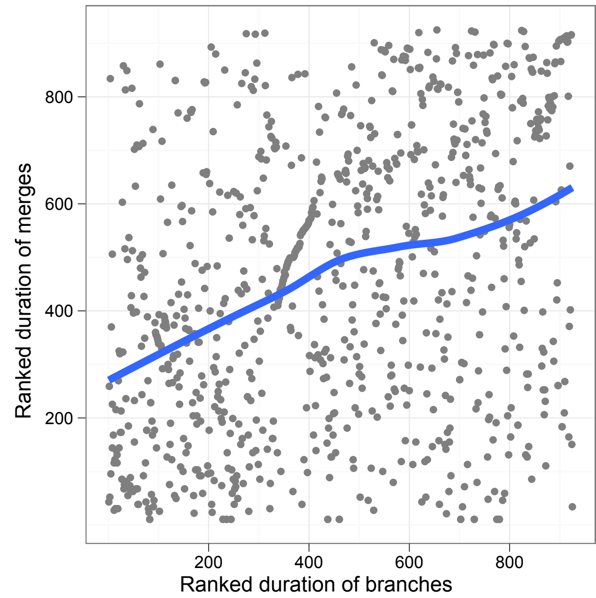

Figure 2: Scatter plot of ranked branch and merge durations

correlation was 0.30, both significant at $p < .001$ indicating that the longer the branch time, implying the size and or complexity of the change, the longer it takes to merge. While statistically significant, the positive correlation between the ranked values must be interpreted with caution: Figure 2 shows several cases where branches with long durations were quick to be merged and vice versa. This can be explained by the nature of the changes in the branch — in some cases, changes are easily made but are complex to merge (e.g., design-related changes). Also the relationship between calendar duration and engineering effort is not always direct — trivial but low priority tasks may take longer to be finished to make way for high priority tasks, there may be dependencies between tasks which increases development time and the like. These scenarios may lengthen the duration of branches and cannot be easily delineated.

Our analysis refutes the claim by the developers from the survey that merges are trivial. While this may be the case for many merges, we have observed several cases where merges cost a lot of time. This makes it clear that there is scope to improve certain practices related to branching and merging so as to reduce associated overheads in the future.

5.2 Geographic Distribution

Initial discussions with two senior employees at Océ led us to speculate that branching occurs in files that are shared by developers located at different geographical sites because physical distance introduces communication and co-ordination challenges [12]. Moreover, we expect files that are shared by co-located developers exclusively have marginal or no branching because they can communicate physically.

We examined the data to investigate whether geographical distances between teams are a cause for heavy branching in shared files. Our findings showed that files are branched irrespective of the fact whether developers are co-located or distributed. In Table 1, we present an anonymized list of

[2] *"Locally weighted regression, or loess, is a procedure for fitting a regression line surface to data through multivariate smoothing: The dependent variable is smoothed as a function of the independent variables in a moving fashion analogous to how moving average is computed for time series." [6]*

Table 1: Branches created within and across sites.

File	Number of Branches	Created in Site A	Site B	Site C
F1.xml	89	47	0	42
F2.xml	60	57	3	0
F3.xml	59	54	5	0
F4.java	30	7	0	23
F5.cpp	28	28	0	0
F6.cpp	24	24	0	0
F7.nsi	22	22	0	0
F8.cpp	20	20	0	0

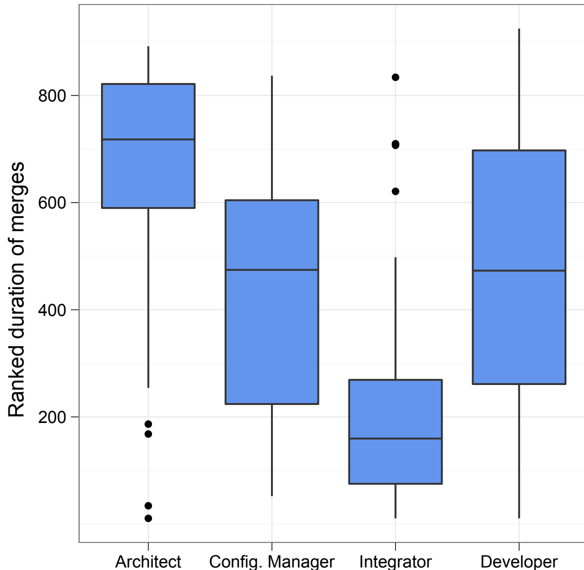

Figure 3: Boxplots of merge durations by role.

files that were often branched. Branches for files `F1-4` have been created across different sites and these files were also shared across the sites in question. But files `F5-8` were only branched in Site A and non-branched revisions were created developers in the same site only. In fact, of the 370 files branched, 326 were branched exclusively within one site.

These results suggest that factors other than geographical distribution influence creation of branches.

5.3 Roles of People

The interviews indicated that people in different roles viewed the overhead of branching and merging differently. While some respondents claimed that merging takes very little time, others stated that occasionally merges can be costly. We noted that the differences in responses often came from respondents with different roles. This led us to examine the data and check whether people in different roles experience overheads from merging files differently. In Figure 3, we plot a boxplot of the distribution of ranked durations of merges across the role of the developer. Marked differences can be observed in the time spent on merging files across roles. Architects seem to take the most time merging files, while integrators are typically quick (except for a handful of outliers).

Table 2: Branching across some file types.

Type	Total files	Branched files	No. of revisions	Branched revisions
apv	1035	20	2355	52
ascii	3507	1	569	2
batch	78	3	259	19
c++	4664	120	11905	579
cfg	40	10	288	46
incl	5590	65	10910	195
java	6586	46	9935	293
makefile	320	7	1023	29
nsi	46	12	648	74
ot_config	8	1	34	2
python	1208	6	4247	12
shsrc	148	5	486	21
txt	948	14	2081	38
xml	1342	33	2580	345
xsl	210	1	332	5

Time spent on merging by programmers/developers scales across a large range which means that some merges are very quick (taking only seconds!), while others take a lot more time. Similarly configuration managers also spend a varied amount of time on merging files. We performed a Kruskal-Wallis[3] test of the roles to check whether their ranked merge durations are significantly different, the test shows that there are significant ($p < .001$) difference of merge-related overheads between the different roles.

The distribution of the ranked durations of merges reflects that certain roles find the automated merge tools useful. For instance, programmers/developers and integrators benefit from using the tool because they typically work on program and configuration files where their full or part ownership can be easily identified. Changes by different people are expected to be in different parts of the code.

Architects, on the other hand, often have to work on code that consolidates from several other developers, which is far more complex and time consuming to perform because of merging cross-cutting changes. This analysis warrants the study of the details of the nature of work done by developers in different roles and devise solutions to further support merging or, even better, avoid branches in the first place.

5.4 Types of Files

In Table 2, we present some of the file types that had undergone branching. The table shows that Océ uses several programming languages including `c++` (`incl` files are header files), `java`, and `python`. Other file types include configuration files (`cfg`), build files (`makefile` and `shsrc`), markup files (`xml` and `xsl`), and testing related files (`apv`).

Column two in the table presents the number of distinct files of the corresponding file type; `c++` and `incl` files together make up for a large majority of files and are followed by `java` files. Column three shows the number of files that are branched at least once. Configuration files (`cfg`) and installation related files (`nsi`) have the highest percentage of branching files — one in four files of these types is likely to be branched. However, note that the number of files for these file types is very low (40 and 46 respectively). On the other hand, 120 `c++`, 65 `incl`, and 46 `java` files can be seen to have been branched.

[3]A statistical test that checks whether the mean of the ranks of a variable across two or more groups is different.

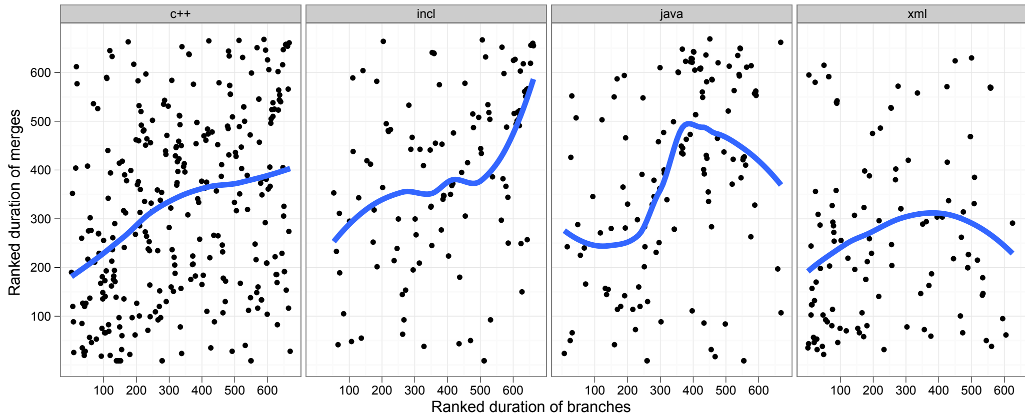

Figure 4: Scatter plot of ranked branch and merge durations across file types

Columns four and five in the table present the number of file revisions for each file type and the number of revisions that are branches respectively. We can see that besides configuration files, `xml` files are heavily branched. The reason for such heaving branching of `xml` files is that these files specify configuration settings and are shared by many developers. Again, the proportion of revisions of program files (`c++`, `incl`, and `java`) is relatively small, but the absolute number of branches is large due to the total number of revisions (approximately 20,000 revisions).

Another investigation is to find out which file types are associated with large merge overheads. The scatter plots of ranked duration of branch times and corresponding ranked merge times across the four file types with most revisions are plotted in Figure 4. The loess smooth in each facet in the figure indicates the general relationship between the branch and merge durations. In the case of `c++` files, the smoother gently progresses upwards indicating that increase in branch time is generally coupled with increase in merge times too. But in the case of `incl` files, the slope of the loess increases dramatically for higher ranks of branch times, which implies that larger changes in `incl` branches take a lot more time to merge than smaller changes. This is not surprising given that the include files typically define the structure of the software system and have wide impact when changes are made to them.

The trend for `java` files varies over the branch duration. The loess smoother for `java` is generally higher than that of `c++`, implying that it takes longer to merge `java` files. Also, the progressive complexity of merging `java` files can be seen by the steep slope in the middle ranges of branch time. But toward the higher branch time ranges, the loess smooth falls. The reasons behind this fall are unclear at the moment. It might be that `java` files that were branched for a long time are of a specific type, and developers developed an effective way to deal with them. This needs further investigation. Lastly, `xml` files are the quickest to merge as suggested by the loess smoother which starts of being comparable to that of the `c++` files, merge time rises with branch time but then the trend turns to the downward side.

We performed an Kruskal-Wallis test to verify whether the ranked merge durations differ across the four different file types. The test indicated a significant difference between the branch-merge time relationship of the file types ($p < .001$). It implies that the amount of branch overheads and merge overheads across file types differ significantly.

5.5 Roles and File Types

Given the above findings that the merge time overheads depend on the merger and the file type, we investigate the effects of the interactions between these two factors. An interaction factor compares a variable's variation within a group across several categories of another group; in our case we compare developer roles across different file types.

In Figure 5, we plot the distribution of ranked merge durations as boxplots across file types and the role of the merger. Architects have been involved in working with all four types of files. They take longest to merge `c++` files, followed by closely related `incl` files. Java files have a wider spread but also have the lowest median of all four files. Lastly, the merge duration for `xml` files is spread across a long range in that merging them can cost time in many cases. It implies that some files can be merged easily and some files can take a long time. Architects have indicated that this could be due to complex changes that affect different modules or sub-systems. On the other hand, the duration of merges for programmers/developers is generally lower than the architects. Programmers/developers are quickest at merging `java` and `xml` files, while `c++` and `incl` files cost them relatively more time. This can be attributed to limited scope of the changes that they are responsible for.

There is only one boxplot for the configuration manager role because they merged only `xml` files. They are faster at merging the files than architects, possibly because of the nature of changes. Integrators are the quickest at merging — they have merged all four file types and for each of them, the distribution of the duration is the lowest.

We performed a Kruskal-Wallis test to verify whether there is a significant effect on the merge time. The test results confirmed that there is a statistically significant relationship ($p < .05$). It shows that the merge duration depends on the role of a software person and the file types that they work on. Since these two factors influence the complexity of merge, potential improvements in the development process can be made if file contents are arranged according to responsibility and ownership.

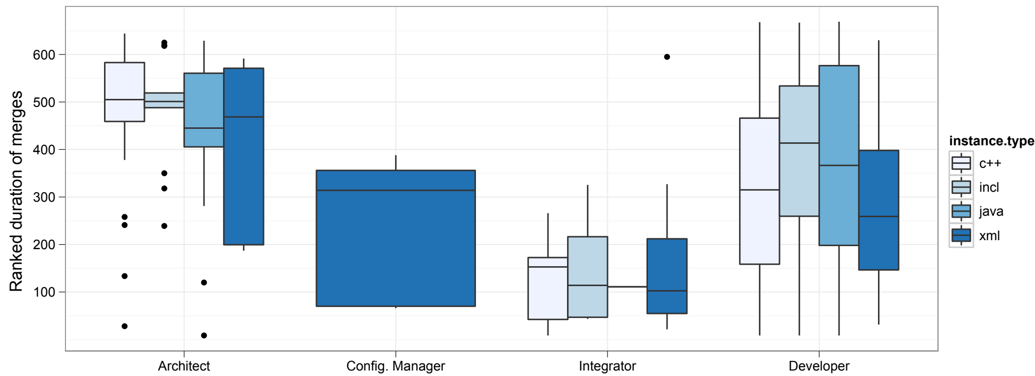

Figure 5: Boxplots of ranked merge durations across roles and file types.

6. RELATED WORK AND DISCUSSION

Software configuration management practices have been proposed to address issues in branching and merging. For instance, Appleton et al. [1] have analysed the branching patterns for parallel software development and proposed several best practices to manage branches better. SCM best practices have also been proposed by Wingerd and Seiwald [15]. SCM best practices are guidelines that support developers in using the SCM. Some of the relevant best practices such as ownership of codelines, branch only when necessary, branch late, propagate early etc. are recommended. From the interviews, we have learned that very few people at Océ are explicitly aware of such practices. Walrad and Strom [14] have described different SCM branching models based on the needs for controlling software release. Buffenbarger and Gruell [5] have studied the branching and merging strategy for parallel software development.

SCM tools such as the one used in Océ have provided much convenience to help developers manage branching and merging. These tools have helped software organizations to manage their software files in an orderly way. However, such convenience may encourage developers to branch indiscriminately, ignoring best practices and malpracticing SCM processes. In this study, we use qualitative and quantitative analysis to understand the developers' behaviour of branching and merging. We discuss the main observations of our study below:

- *Agile development practices has supported branching and merging at Océ.* The survey clearly reflected that developers at Océ branch files opportunistically. This may partly be a consequence of the workflow at the company, i.e., adhering to agile principles where tasks are planned for a 2 week long sprint and several developers may have to share and edit files simultaneously. So it is natural that developers do not hesitate to create branches in order to ensure that their own deadlines are met. Having said that, parts of files at Océ may implicitly belong to certain developers so as to avoid cross-cutting changes (i.e., no conflicts by making changes in the same part of the code) and thereby reducing the burden arising from the merging of files.
- *Automated merge support and delineation of responsibilities can reduce overhead.* Both the survey and quantitative analysis show that the automated tool support to merge files is valuable. As we have seen, many merges are short in duration, which may be possible because of localised changes. The developer's work in such cases is reduced to only confirming the merge proposed by the tool. This is a big advantage over having to do all merges manually. The automated merges have in some ways shaped the developer responsibilities of files such that editing conflicts can be minimised and the benefits of automated merging can be maximised.
- *Developer role and file type together influence the overheads of merging files.* The quantitative analysis shows that developers in different roles experience different merging overheads. We noticed that architects take most time to merge file changes for all types of files. This can be explained by the fact that they typically integrate files that are more complex, and those files can be worked on by several teams. Thus the architects are not able to rely on automated merging as much as the developers. Due to the complexity of the merges, architects need to manually merge and check that the results of the merged files remain consistent. Developers, on the other hand, take lesser time than architects to merge changes because the changes made by developers are typically local, to a single file or a distinct part of a file, and hence, automated change merges can be used without complications.

  Amongst the different roles, we see some differences in merge durations across file types (Figure 5). `xml` file changes appear to be quickest to merge across the different developers, while program file changes take longer. The duration of merging of `java` and `c++` files between architects and developers differ. Architects take longer to merge changes from `c++` than `java`, and the reverse is true for developers. It implies that the complexity of merge for each file type by different roles are different depending on what they do. Hence, any measures to reduce merge overheads should take into account both the type of the file and how different people work with those files.

An important lesson learnt from this work is that there are benefits in structuring both program and other files to

reflect not only the architecture of the system, but also the team structure. One of the respondents from the survey replied: *"Some organization alignment is essential (allocation of modules to owners) so that communication about prevention of unnecessary or undesirable branching is possible."* Dividing ownership of files or its parts can go a long way in allowing parallization, and facilitate developers to branch with negligible impacts on merging changes. This idea is validated by the developers' comments (Section 4.4, Q8.). For files that are not arranged according to the developers' ownership, they have to be split up into several files to eliminate the need to perform complex merging. During the time that this work is conducted, system integrators have split up several heavily branched `xml` files to avoid merge-related issues, and since then these files have undergone only marginal branching, and merging changes became a lot easier.

7. THREATS TO VALIDITY

We have studied a few projects only within Océ. Since other projects within Océ work in a similar agile fashion, and rotation of people between projects is quite common, we are confident that our findings hold true for all of the company. We do not yet know whether our findings hold for other companies that use agile practices because their workflow and tools may differ to those used at Océ.

Respondents for our survey were selected based on their branching/merging behaviour, not on their roles. If we would have selected developers balanced in number across roles, we might have collected more evidence about situations where merging is difficult and undesired. We did not initially suspect that role may be a factor, and then it was too late to re-survey people.

8. CONCLUSION

The agile development environment at Océ has encouraged developers to focus on software delivery. A side effect is that developers would want to edit files when they need them. This creates a potential issue that files managed by a SCM tool can be branched heavily. Typically SCM best practices such as "branch only when necessary, branch late, propagating early and often" do not seem to be followed by the developers.

In this study, we try to understand the implications of such branching on the cost of merging changes. We have found that (a) the roles of the branchers and the mergers, and (b) the type of files that they work with dictate the cost of merging. Software developers who work on files that have distinct ownership can branch freely without having an impact on the merge. These kind of merges typically takes very little time because the SCM tool can perform the merge effectively. Architects who work on branched files typically are more careful with branching because the files that they work on involve multiple parties and the files are typically more complex in nature. They would not branch casually, they would check with others before branching and merging. SCM best practices can help this group of people.

Configuration files such as `xml` files also require distinct ownerships to avoid complications in merging. Before the study, some of these files were branched heavily and the merge time was quite high. Since then, they have been split into smaller files with distinct ownerships. Branching of these files has thereafter been reduced substantially.

This study has provided insights to show that the use of SCM tools and SCM best practices are not sufficient to share files in an agile development environment. The contents of the files that are shared must be aligned with the responsibilities of the primary owners of those files. In that way, conflicts of branching and merging files can be minimised.

Acknowledgements: This research has been partially sponsored by the Dutch "Regeling Kenniswerkers" Stephenson (KWR09164): Architecture knowledge sharing practices in software product lines for print systems.

9. REFERENCES

[1] B. Appleton, S. Berczuk, R. Cabrera, and R. Orenstein. Streamed lines: Branching patterns for parallel software development. In *Pattern Languages of Programs*, 1998.

[2] U. Asklund and L. Bendix. A study of configuration management in open source software projects. *IEE Procs. Software*, 149(1):40–46, 2002.

[3] S. Berczuk and B. Appleton. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Addison-Wesley, 2002.

[4] C. Bird, P. Rigby, E. Barr, D. Hamilton, D. German, and P. Devanbu. The promises and perils of mining git. In *Int. Working Conf. on Mining Software Repositories*, pages 1–10. IEEE, 2009.

[5] J. Buffenbarger and K. Gruell. A branching/merging strategy for parallel software development. In *Software Configuration Management*, pages 86–99, 1999.

[6] W. S. Cleveland and S. J. Devlin. Locally weighted regression: An approach to regression analysis by local fitting. *J. of the American Statistical Association*, 83(403):596–610, 1988.

[7] J. Estublier. Software configuration management: a roadmap. In *ICSE - Future of SE Track*, pages 279–289, 2000.

[8] J. Estublier, D. Leblang, A. Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber. Impact of software engineering research on the practice of software configuration management. *TOESM*, 14(4):430, 2005.

[9] T. Mens. A state-of-the-art survey on software merging. *TSE*, 28(5):449–462, 2002.

[10] N. Pala Er and C. Erbaş. Aligning software configuration management with governance structures. In *Workshop on Software Development Governance*, pages 1–8. ACM, 2010.

[11] D. E. Perry, H. P. Siy, and L. G. Votta. Parallel changes in large-scale software development: an observational case study. *TOESM*, 10(3):308, 2001.

[12] D. Smite, C. Wohlin, T. Gorschek, and R. Feldt. Empirical evidence in global software engineering: A systematic review. *Empirical Software Engineering*, (1):91–118, Feb.

[13] G. L. Thione and D. E. Perry. Parallel changes: Detecting semantic interferences. volume 1, pages 47–56, 2005.

[14] C. Walrad and D. Strom. The importance of branching models in SCM. *IEEE Computer*, 35(9):31–38, 2002.

[15] L. Wingerd and C. Seiwald. High-level best practices in software configuration management. *System Configuration Management*, pages 57–66, 1998.