A Qualitative User Study on Preemptive Conflict Detection

Lile Hattori, Michele Lanza, Marco D'Ambros
REVEAL @ Faculty of Informatics - University of Lugano, Switzerland

***Abstract*—Preemptive conflict detection is the act of detecting a potential merge conflict at an earlier stage than at check in time, and informing the involved developers about it. Researchers have proposed a number of tools and techniques to detect potential merge conflicts. However, barely any study has been conducted to investigate whether the adoption of such tools and techniques brings benefits to developers.**

We have conducted a qualitative user study to understand how developers behave when dealing with merging and how this behavior changes when they are exposed to preemptive conflict detection. We report on the analysis of the data collected in the user study, and provide an in-depth discussion on the findings derived from it.

I. Introduction

Teamwork plays an important role for a successful delivery of a software system [1]. An important aspect of team collaboration is awareness, defined as an understanding of the activity of others that provides a context for one's activity [2]. In the context of software development, awareness is the means by which team members can become aware of the work of others that potentially impacts their own work [3].

In a collocated team, awareness is mainly obtained through human interactions, such as meetings, and informal conversations [4], [5]. In a distributed team, however, the distance among developers or teams is a barrier to informal interactions, and the awareness of the team's activity is consequently lower than in collocated teams.

Recently, there has been a significant effort in the software configuration management (SCM) community to increase awareness of distributed teams by supporting coordination across multiple developers working in parallel [6], [7], [8]. These approaches to promote *workspace awareness* preempt merge conflicts on SCM systems by detecting in real time concurrent modification to software artifacts that are potentially conflicting: concurrent changes that are likely to cause merge conflicts at check in time.

A limited number of studies [6], [9], [8] have been conducted to evaluate whether the adoption of tools to promote workspace awareness are beneficial to developers. Their initial findings suggest that, when preemptive conflict detection is introduced, (i) the frequency of communication increases, (ii) there is a reduction in overlapping work, and (iii) an increase in the detection and resolution of conflicts takes place.

However, fundamental questions concerning these approaches are seldom discussed: Were the changes observed in these studies beneficial to developers? Did the developers' strategies to deal with merging change? Is the information being delivered disrupting? Would they prefer to get the information in a different way?

We have conducted a qualitative user study to understand how developers behave when dealing with merging, how this behavior changes when preemptive conflict detection is present, and whether this change is perceived as beneficial. We also investigated how developers prefer this information to be delivered, by exposing them to two different ways of visualizing emerging conflicts and collecting their opinions.

In our experiment, we collected data from recorded observations, interviews, and questionnaires, and analyzed the data iteratively to extract the most important findings. When developers are exposed to preemptive conflict detection, we have observed a decrease on the frequency and an increase on the depth of communication, as well as a change on the strategies to merge code. These changes are beneficial, since they help developers to successfully deal with merging, in contrast with the struggle to merge code without the information on emerging conflicts.

The contributions of this paper can be summarized as: (1) *an approach to detect and visualize emerging conflicts*, including a conflict detection algorithm and two different ways to view conflicts in the IDE; (2) *a report on the design and operation of a qualitative user study* to understand how developers behave when dealing with merge and how this behavior changes when preemptive conflict detection is present; (3) *a detailed analysis of the results*, obtained through the collection of multiple data sources (questionnaires, observation, recorded screencasts, interviews, and documentation), that shows beneficial changes on the behavior of developers when they are exposed to preemptive conflict detection.

II. Related Work

Grinter conducted a field study that investigated developers' coordination strategy [10], which is composed of field observations and semi-structured interviews conducted with two development teams of distinct software companies. Grinter observed that developers are constantly faced with a dilemma: on the one hand they want to finish their work first to avoid merging; on the other hand, they want to produce quality code. Grinter also observed that even experienced developers face problems merging code.

The second study was an eight-weeks observation and note taking of the coordination practices of a NASA software team [1]: de Souza observed that developers tend to speed up to finish their activities earlier to avoid merging. Another observation is that developers perform partial check-ins of files that are frequently changed. Some developers, however, tend to hold their check-ins until the end of a work day.

In a laboratory study, Sarma *et al.* [8] observed that participants who used their workspace awareness tool, Palantìr, detected and resolved more conflicts than those with no conflict detection tool. They also coordinated more than those not using it. Some coordination strategies were observed: check in partial changes, rush to complete their changes so they can check-in first, use placeholders, skip their current task, and chat to informally coordinate their tasks. This evaluation shows promising results, however it is important to further investigate whether the trade-off between the benefits and the cost in added coordination pays off.

Biehl *et al.* [6] evaluated the impact of change awareness and conflict detection. They conducted an observational study of a development team of 6 people before and after the introduction of the tool. The most important results were the increase in communication due to increased awareness, and the reduction in overlapping work. Dewan and Hedge [9] evaluated the usefulness of CollabVS's mechanism to detect and fix conflicts at editing time. The tool showed to be useful to increase the developer's ability to resolve conflicts with an increase in communication for resolving indirect conflicts.

Motivation for the present work: Our own user study complements the studies performed so far. It studies the behavior of developers when performing SCM operations, relating the different strategies with the developers' experience. It also investigates how developers change their behavior when exposed to preemptive conflict detection, and whether this change is beneficial for them or not. Lastly, we collected developers' opinion of how to present the information of emerging conflicts without disturbing their main focus, *i.e.,* producing quality code.

III. Preemptive Conflict Detection

A. Conflict Detection Algorithm

Our conflict detection algorithm resides on the server of Syde [11], the tool we devised to help developers to collaborate. Syde is a client-server application that tracks the fine-grained changes performed by developers on a shared software project in Eclipse.

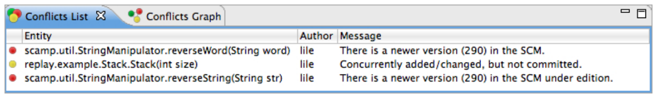
Our algorithm detects structural conflicts [12] related to the changes Syde tracks. It detects both direct and indirect conflicts. Direct conflicts refer to changes concurrently made to the same program artifacts; indirect conflicts refer to changes made on an artifact that can impact an interdependent artifact. Indirect conflicts can be further classified into syntactic or semantic. We have addressed indirect conflicts due to syntactic changes.

The created conflicts are classified as *yellow* when there is a structural conflict between two entities, but none was checked in the SCM system; and as *red* when there are structural differences between two entities, and one of them has been checked in the SCM system. Once created, a conflict can change severity (*e.g.,* yellow become red), or be resolved.

B. Conflicts Plug-in

Once potential merge conflicts are detected by the algorithm on the server, they are presented to the involved developers in their IDE in two different ways:

- *List View* shows all potential conflicts as a list.
- *Graph View* shows the classes in a graph and potential conflicts as a color layer on top of each node.



	Entity	Author	Message
🔴	scamp.util.StringManipulator.reverseWord(String word)	lile	There is a newer version (290) in the SCM.
🟡	replay.example.Stack.Stack(int size)	lile	Concurrently added/changed, but not committed.
🔴	scamp.util.StringManipulator.reverseString(String str)	lile	There is a newer version (290) in the SCM under edition.

Figure 1. List View showing potential conflicts: two red, and one yellow

List View. Similarly to the ones of previous tools [7], [8], it shows all the potential conflicts that a developer might be involved in (see Figure 1). It shows the method or class where the conflict is, the other developer involved, a description of the conflict, and the status (yellow/red). If a developer clicks on a conflict, the file containing the conflict opens with the involved class or method in focus.

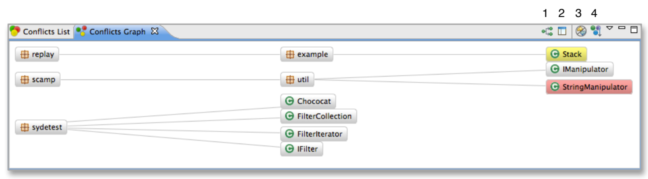


Figure 2. Graph View: Potential conflicts in Stack & StringManipulator.

Graph View. It shows a graph representation of the system, with the packages and classes as nodes and containment or call dependency as edges (see Figure 2). It notifies developers about emerging conflicts by coloring the class nodes. The edges ease the understanding of indirect conflicts or the identification of nodes potentially affected by a ripple effect. By hovering over a node one sees where the conflict is located, who the involved developer is, and read the conflict's description.

IV. User Study Design

We want to conduct a qualitative study on how developers behave when dealing with merging code and how their behavior changes when they are exposed to preemptive conflict detection. Moreover, we want to investigate how developers prefer to receive information on emerging conflicts. To this aim, we formulate the following research questions:

RQ1 How do developers behave when they have to merge code and resolve conflicts?

RQ2 How does the behavior of developers change when information of emerging conflicts is present?

RQ3 How do developers perceive different approaches to deliver information on emerging conflicts?

We designed a user study in a laboratory setting to simulate developers working in parallel to closely observe their behavior. The user study consisted in a programming assignment with three tasks performed by two developers simulating parallel development. Each assignment is designed to cause a merge conflict. To simulate a distributed environment, developers are forbidden to talk or to see each other's screen. All the communication has to be done over instant messaging (IM). The participants are given an Eclipse installation with the necessary tools (Subversive, SVN connectors, Syde's plug-ins) to perform the assignment and an IM client for communication.

A. Data Collection

Following the guidelines of Creswell [13] and Barbour [14] to use mixed data collection methods for a better interpretation of our findings, we collected the following types of data:

- **Questionnaires:** A screening questionnaire to collect the participants' experience level and use it to characterize the participants and take their knowledge into consideration in the data analysis; a debriefing questionnaire, after the assignment, to collect participants' immediate feedback on the experiment.
- **Observation:** We recorded each participant's interaction with the tools to later perform the observation on the screen cast. To properly analyze the screen cast, we developed a codebook, containing 15 codes that were classified into four categories (communication, interaction with SCM, interaction with Syde, and testing), to annotate the videos. We coded a total of 918 minutes (ca. 15 hours) of screen casts and used the resulting annotations for qualitative and quantitative analyses.
- **Interviews:** At the end of each session, we conducted a semi-structured interview with the two participants at the same time to collect feedback from them. There were 12 questions that served as a guide, however, given the semi-structured method, each interview had different set of questions that were asked according to the answers participants gave to questions previously asked.
- **Documentation:** The chat logs, and the list of changes and conflicts generated during the assignment were collected and used in the data analysis when some of the other collected data was missing.

The handouts of the tasks, the questionnaires and the participants' answers, the codebook, and the interview questions are reported at [15].

B. Object System & Tasks

The system we chose as object of our experiment is *Checkstyle* 5.3, which consists of 341 classes distributed across 22 packages, for a total of 46 KLOCs. Our choice was motivated by the following factors: Checkstyle's size allows for performing a user study session, yet being representative of real life programs. Moreover, it has been used in previous experiments [16], [17], [18].

The experiment's assignment was composed of three coding tasks to be done by two participants in collaboration. Each participant had a different, but complementary, set of tasks. For a task to be considered finished, each participant had to code what the task was requesting, coordinate with his pair, check in his changes, update the code with the changes done by his pair, and make sure all the tests related to the task pass. Each task was accompanied by instructions to prepare for it, which contained the tools and views that participants were allowed (or not) to use.

C. Operation & Pilot Studies

The user study is composed of runs, in which two participants change the object system in parallel to solve the tasks. Each run includes a training session of approximately 15 minutes and one experimental session. The *training session* consists in a tutorial on the views, a hands-on session with a toy system, in which the participants can experiment with the views, and a warm-up task to allow the participants to get familiar with the object system. The *experimental session* is composed of three programming tasks with unlimited time to solve them. The first task does not involve preemptive conflict detection, while in the second and third either the list or the graph view is used. There were 6 experimental runs for a total of 12 participants.

Before these runs, we ran two pilot studies. In the first one, we tried to simulate the second person of a pair of participants, however it was unfeasible to take into account all the factors that influence a person's behavior. We redesigned the assignment by creating a set of complementary tasks for two participants, and ran a second pilot study with two people. This time, the merge conflicts appeared when we planned to, and the participants, located in different continents, were able to communicate over chat and to solve the tasks collaboratively.

V. Data Analysis

In the following we discuss the research questions based on the analysis of the data collected from the six experimental runs. The complete report on the events that happened with a single individual or within a single run can be found in [15]. Here we analyze the changes in communication and coordination strategies. We noticed that the participants are naturally split into two distinct groups:

- *Beginners:* The participants of three runs are MSc students with no experience with developing industrial

size systems. Though they reported to have experience with SCM systems, some of them clearly have little experience, and struggled with certain SVN operations or using them within Eclipse.
- *Advanced:* The participants of the other three runs are PhD students with at least one year of experience developing industrial size systems. Most of them reported to have previously worked in the industry, thus having practical experience in the field.

A. *RQ1: How do developers behave when they have to merge code and resolve conflicts?*

The answer of this question is derived from the data collected for Task 1, which involved the improvement of a method for one developer, and the refactoring of the code of the same method into a second one for the other developer.

The observations of how developers behave when they have to merge code can be summarized as follows:

- *Beginners adopt different naïve strategies.* Developers who are inexperienced with SVN tend to use different naïve strategies to merge code. These strategies seem to derive from strategies taken when using SVN through the command line. They struggle to understand how to properly merge the code, sometimes erasing their own changes, or erasing the changes of others.
- *Advanced developers behave similarly, but struggle with merge.* They usually follow the strategy of synchronizing the code, inspecting the conflicting classes to understand their differences, merging the code by resolving the conflicts, and checking the merged code in. However, in most cases, they had difficulties to understand how to properly merge the code, introducing errors that broke the tests. This confirms Grinter's observation that even experienced developers face problems when merging code [10].
- *Developers communicate after the first check-in.* With one exception, all developers started to communicate after the first check in or after the first failed attempt to check in.
- *Communication is kept shallow.* In most cases, communication is kept at the bare necessity. Developers tend to communicate only when it is really necessary to continue the coding task, otherwise they tend to code only.
- *A few developers have deeper communication.* A few developers communicated more, trying to explain the changes they introduced or to understand the changes introduced by the others. Some even shared code snippets over the IM.
- *Some developers rush to check in first to avoid dealing with merge.* This behavior we observed confirms previous findings that developers try to avoid dealing with merge by commiting changes before their colleagues [1], [10].

B. *RQ2: How does the behavior of developers change when information of emerging conflicts is present?*

The answer to this question is derived from the data collected for Task 2 and Task 3. In these tasks developers had the aid of preemptive conflict detection to alert them of emerging conflicts in real time.

The observations of how the developers' behavior changed when exposed with notifications of emerging conflicts can be summarized as follows:

- *Developers start to communicate earlier and have meaningful communication.* With the exception of one run, all developers start to communicate before check-in time. Most of them also put more effort in explaining to their counterpart the changes they introduced.
- *Developers coordinate more effectively.* Two new coordination strategies are adopted by developers in different runs: to discuss and determine who should check in first and who should merge (giving preference to the most experiences developer to merge); and to split the check-ins into smaller ones to reduce the complexity of the merge.
- *Beginners show a slower change of behavior.* They have some difficulties to understand the concept of preemptive conflict detection. In consequence, they have a slower change of behavior. In one case the gradual change of behavior leads the participants to successfully dealing with merging.
- *Advanced developers succeed in merging code.* Differently from what happens in Task 1, developers manage to merge code successfully. We attribute this success to a higher level of awareness, and an improved coordination.

C. *RQ3: How do developers perceive different approaches to deliver information on emerging conflicts?*

Most participants prefer the list (eight of them), and the strongest reason is that the list presents the important information in a direct and condensed manner. Thus, developers only need to look at the view to get all the information they need about an emerging conflict. Only three of the participants prefer the graph, with different reasons for their choice.

Suggestions on other ways to visualize emerging conflicts: Many participants had the same suggestion, which is to show emerging conflicts directly in the Java editor. Even though this is a more intrusive approach, developers showed a strong preference for it. Having an enable/disable option should be enough to let a developer disable it if he feels disturbed by the notifications. Showing information on emerging conflicts only in the Java editor would prevent developers from receiving overall information on the system. Hence, the views are an alternative to show the global view of conflicts, providing complementary information.

VI. Concluding Remarks

We reported on the behavior of developers when dealing with merging code without the help of preemptive conflict detection. Our findings confirm those from previous studies: developers, even the experienced ones, struggle with merging [10]; and some developers tend to rush to check in first to avoid dealing with merging [1], [10]. Additional findings are: developers only started to communicate, and consequently to coordinate, after the first (attempted) check-in; and in most cases the communication remained shallow, with only a few developers putting effort in explaining to their counterpart the performed changes. In terms of conflict resolution, experienced developers showed similar behavior amongst them, while beginners used different, and mostly naïve, strategies.

We found significant changes on developers' behavior after the introduction of preemptive conflict detection, similar to the changes observed by Sarma *et al.* [8]. Developers started to communicate earlier, usually right after the first information on emerging conflicts appeared. The early and more meaningful communication helped them to coordinate better by adopting different strategies than what they would normally do. We observed two new strategies: to discuss and decide upfront who would merge the conflicting code; and to break the check-ins into smaller ones aiming at reducing the complexity of the merge.

These behavior changes proved to be beneficial, given that most of the developers succeeded in merging when preemptive conflict detection was present, in contrast with previous struggle when this information was not available. Moreover, the participants' perception on emerging conflicts was that they could indeed be helpful in their work environment.

Lastly, we argue that different ways to visualize this information are complementary to one another, because developers have different preferences, and the fact that they can choose among different options might help them in the adoption of preemptive conflict detection. Therefore, there is potential for preemptive conflict detection to be adopted by practitioners, although more research towards effective ways to deliver this information in IDEs should be conducted.

Acknowledgements. We thank the participants of our study. We also thank Alberto Bacchelli for the feedback on early versions of this paper. Hattori is supported by the Swiss Science foundation through the project "GSync" (SNF Project No. 129496).

References

[1] C. R. B. de Souza, D. Redmiles, and P. Dourish, "Breaking the code, moving between private and public work in collaborative software development," in *Proceedings of GROUP 2003 (International ACM SIGGROUP Conference on Supporting Group Work)*. ACM Press, 2003, pp. 105–114.

[2] P. Dourish and V. Bellotti, "Awareness and coordination in shared workspaces," in *Proceedings of CSCW 1992 (ACM Conference on Computer-supported Cooperative Work)*. ACM Press, 1992, pp. 107–114.

[3] D. Damian, L. Izquierdo, J. Singer, and I. Kwan, "Awareness in the wild: Why communication breakdowns occur," in *Proceedings of the ICGSE 2007 (International Conference on Global Software Engineering)*. IEEE Computer Society, 2007, pp. 81–90.

[4] J. Herbsleb, A. Mockus, T. Finholt, and R. Grinter, "Distance, dependencies, and delay in a global collaboration," in *Proceedings of CSCW 2000 (ACM Conference on Computer Supported Cooperative Work)*. ACM Press, 2000, pp. 319–328.

[5] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceedings of ICSE 2006 (28th ACM International Conference on Software Engineering)*. ACM, 2006, pp. 492–501.

[6] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson, "Fastdash: a visual dashboard for fostering awareness in software teams," in *Proceedings of CHI 2007 (25th SIGCHI Conference on Human Factors in Computing Systems)*. ACM, 2007, pp. 1313–1322.

[7] R. Hegde and P. Dewan, "Connecting programming environments to support ad-hoc collaboration," in *Proceedings of ASE 2008 (23rd IEEE/ACM International Conference on Automated Software Engineering)*. IEEE CS Press, 2008, pp. 178–187.

[8] A. Sarma, D. Redmiles, and A. V. der Hoek, "Palantir: Early detection of development conflicts arising from parallel code changes," *IEEE Transactions on Software Engineering*, vol. 99, no. PrePrints, 2011.

[9] P. Dewan and R. Hegde, "Semi-synchronous conflict detection and resolution in asynchronous software development," in *Proceedings of ECSCW 2007 (the 10th European Conference on Computer Supported Cooperative Work)*. Springer, 2007, pp. 24–28.

[10] R. Grinter, "Supporting articulation work using software configuration management systems," *Computer Supported Cooperative Work*, vol. 5, no. 4, pp. 447–465, 1996.

[11] L. Hattori and M. Lanza, "Syde: A tool for collaborative software development," in *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, 2010, pp. 235–238.

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

[13] J. Creswell, *Qualitative Inquiry & Research Design*, 2nd ed. Sage, 2007.

[14] R. Barbour, *Introducing Qualitative Research*. Sage, 2008.

[15] L. Hattori, M. Lanza, and M. DAmbros, "A qualitative analysis of preemptive conflict detection," University of Lugano, Tech. Rep. 2011/05, Sep. 2011.

[16] B. Cornelissen, A. Zaidman, and A. van Deursen, "A controlled experiment for program comprehension through trace visualization," *IEEE Transactions on Software Engineering*, vol. 99, 2010.

[17] A. Guzzi, L. Hattori, M. Lanza, M. Pinzger, and A. van Deursen, "Collective code bookmarks for program comprehension," in *Proceedings of ICPC 2011 (19th IEEE International Conference on Program Comprehension))*, 2011, pp. 101–110.

[18] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Software Engineering*, vol. 16, pp. 325–364, 2011.