Speculative Identification of Merge Conflicts and Non-Conflicts

University of Washington Technical Report UW-CSE-10-03-01

Yuriy Brun, Reid Holmes, Michael D. Ernst, David Notkin
Computer Science & Engineering
University of Washington
Seattle, WA 98195-2350, USA
{brun,rtholmes,mernst,notkin}@cs.washington.edu

ABSTRACT

Most software is built by multiple people, and a version control system integrates evolving individual contributions into a whole. Every engineer makes decisions about when to incorporate other team members' changes, and when to share changes with other team members. Sometimes, an engineer performs these tasks too early, and in other cases performs them too late. In this paper we address several questions to determine if there are enough situations in practice where an individual could benefit from explicit knowledge about the relationship between their view of the software with respect to other views of the software. In particular, we speculate (in principle) at each moment in time about whether unrecognized conflicts with teammates exist and whether there are unnoticed opportunities for straightforward merging among teammates.

To determine whether there are sufficient potential opportunities — needed to justify the design, implementation, and evaluation of a speculative tool — we analyze existing source code repositories. Across several open-source projects, we compute and report results including how long conflicts persist before they are resolved (a mean of 9.8 days) and how long opportunities for a non-conflicting textual merge persist (a mean of 11 days). In addition, for one of the projects, we compare the persistence of textual conflicts vs. compilation conflicts vs. testing conflicts. Our data show that there is ample opportunity to benefit from speculative version control, justifying a tool design and implementation effort.

1. MOTIVATION

A software engineer continually applies commands such as "lift this method to its superclass," "incorporate my changes with changes made by my team," etc. The engineer generally has an expectation, but no certainty, of the effects and consequences. For example, applying a command may cause merge conflicts, compilation failures, or changes in test results, but the engineer only learns of such consequences afterwards. We propose that an integrated development environment (IDE) can speculatively compute, in the background, the consequences of potential commands an engineer may apply. We expect that this added, precise information can help the engineer make better decisions.

This paper focuses on a set of speculations specific to collaborative development. Most software is built by multiple people, and a version control system integrates evolving individual contributions into a whole. Each individual's work deviates in varying degrees, over time, with respect to shared views of the software. Every engineer makes decisions about when to incorporate other team members' changes, and when to share changes with other team members. Sometimes, an engineer performs these tasks too early, and in other cases, performs them too late. We posit that there exists information that could allow engineers to make better decisions. Our research is a search for that information.

Consider the following illustrative scenario. Melinda and Bill are working on Features 1 and 2, respectively, of an operating system. Over the course of a week, Melinda makes some changes in her copy of the code; Bill does likewise in his own copy. When all the Feature 1 tests pass, Melinda shares her changes to the code repository. (We will define more formally what we mean by "sharing changes" in Section 4.1.) A day later, Bill verifies that the Feature 2 tests pass on his copy of the code, and shares his changes to the central repository. There are no textual conflicts between Bill's changes and those of any other team member. However, the following morning, the system test suite indicates a failure caused by an interaction among the features. The developers must review each change they made, remembering the reasoning and rationale, to discover and understand the unintended interaction of their features that broke the regression tests, and to correct the problem. It would have been better for Bill to be notified, before he shared his changes to the central repository, that this operation would cause the tests to fail. Even better, Bill could have been notified of the problem as soon as Melinda shared her changes, even before he considered sharing his changes. Best of all, Bill and Melinda could both have been notified as soon as they made local commits whose combination destabilized the system, even before either had shared those changes to the central repository. At that time, Bill and Melinda are engrossed in the details of their changes and can resolve them more effectively and quickly.

As a variation on the above scenario, suppose that Bill had updated his copy of the code and run the full system tests before sharing his changes. In this case, Bill and Melinda would be in the same situation as before, with the same hard task of remembering and reconciling their work, after the fact. It is often infeasible or undesirable for a developer to delay his work to run full system tests.

As a third scenario, suppose that Paul, who is working on Feature 3, is worried about the possibility of a long-lived unnoticed semantic conflict. He knows that the longer a problem lingers, the harder it is to fix, so he wishes to nip any such problem in the bud. Therefore, he regularly updates his copy of the code to incorporate Melinda's and Bill's changes. In general, this works well. But occasionally, it causes Paul's system to break, and he has to waste a lot of time resolving conflicts — sometimes repeatedly — in order to get back to his own work. It would be better if Paul knew which commits in the central repository he could safely incorporate into his own work. He could choose to defer the others until his feature was complete.

To make progress towards the collective goal, each engineer vacillates away from and back towards the master version as well as the copies of others in their teams. In the face of imprecise information, each individual (1) may underestimate how far they have deviated from others, likely causing costly merges, undos, and repeated work, or (2) may overestimate how far they have deviated from others, scaring them from incorporating others' changes earlier.

We propose that an IDE can speculatively perform version control operations (or any other software development action), record the outcomes, and then make this information unobtrusively available to developers. For example, when Melinda and Bill are working on their own changes, the IDE can speculate that they will need to merge their work with the master version or with each other; the IDE can perform such merges, compile the code, and run the tests, all in the background and in a separate workspace to prevent affecting the developers' ongoing work. Knowing the consequences can help a developer to make better decisions, such as resolving conflicts before they deepen or avoiding getting distracted by a merge task.

This paper answers the question: do sufficient opportunities exist in practice that would allow a speculative mechanism like the one we have described to help programmers better manage their integration activities? Specifically, we analyze the space of software development projects and answer several research questions about whether or not IDEs can properly speculate about potential actions in a collaborative environment and bring information that is unavailable today to the attention of the engineers, reducing the time to conflict discovery and increasing engineer confidence in safe merges.

In our analysis of the histories of subsets of eight real-world open-source programs, each with up to 1.5 million lines of code and up to 100 developers, we found ample opportunities for an IDE to deliver pertinent information about speculative merges to the developer. First, we found that branching and merging are widely used in today's software development. Second, we found that 17% of merges result in textual conflicts that require human intervention to resolve, presenting opportunities for the IDE to both (1) warn the developer of the conflict when it first happens and (2) increase the developer's confidence that a merge will not result in a conflict, allowing developers to share more often and diverge less from each other in their development. Third, we found that textually conflicting branches persisted on average 10 days, and sometimes as long as year, before being resolved. Similarly, textually non-conflicting branches persisted for an averge of 11 days, and up to four months, without work being shared. Further, when we applied deeper conflict analysis and examined merges that had no textual conflicts, we found that an additional 19% (for a total of 38%) of the merges contained compilation, halting, or behavioral conflicts that developers would normally learn about even later, while a speculating IDE could have predictied these conflicts and informed the developers of their potential at a much earlier time.

The remainder of this paper is structured as follows. First, Section 2 outlines the research questions we aim to answer. Next, Section 3 describes some anecdotal evidence that industrial managers have a strong interest in an IDE with the capabilities we describe here. Then, Sections 4 and 5 analyze eight real-world programs to answer our target research questions. Finally, Section 6 describes how our work fits within the related work in the field and Section 7 summarizes our contributions.

2. RESEARCH QUESTIONS

We believe that by speculating a collaborating engineer's potential future actions, an IDE can provide useful information about the engineer's deviation from the rest of the team. The primary contribution of this paper is an evaluation of whether such information exists. Only if the answer is affirmative does it make sense to build a tool and perform a user study to determine whether the information is useful to an engineer.

Thus, we pose the following research questions:

RQ-1 How often do conflicts occur?

RQ-2 How long do conflicts persist?

RQ-3 How much earlier could an IDE discover potential conflicts with other developers than they are being resolved by engineers today?

RQ-4 How often could textually safe merges take place but do not?

RQ-5 Can we capture data about other types of conflicts that can occur in practice?

RQ-6 Are there high-order conflicts, such as ones that manifest themselves through regression tests, that do not manifest themselves in textual merge conflicts? That is, is there any benefit to and IDE performing high-order analyses?

3. AN INDUSTRIAL CORROBORATION

A private communication with (a friend who is) an industrial development manager corroborated the basic scenarios sketched in Section 1. The manager runs a software project with a local team of 12 developers in North America and with two remote teams of eight and of 17 developers, each in India. He is increasingly concerned about keeping the source code produced by his remote teams synchronized with his local team's code; specifically, he is concerned that the remote teams are frequently deviating too far from the local team's branches, causing time consuming merges and wasted effort.

Given this context and absent any knowledge of our work, he said (via instant messaging):

```
[We'd like an extension of] continuous in-
tegration, but providing additional detail
before you commit.  In Eclipse parlance, each
developer's IDE (remote or otherwise) would
maintain a shadow workspace, and using net-
worked updates derived from the local his-
tory deltas of all members of the team, you
would get instant feedback if someone is break-
ing the build or at least mucking with the
```

```
files you are thinking about changing or com-
mitting changes on.  Why wait until you try
to merge?
```

While each remote team was able to keep its own code consistent, they would stray from the two other teams remote to them. Since the remote teams knew merging was an expensive process, they would only do it once or twice during their two-week code sprints. The downside to this, of course, was that the project's continuous integration system was only compiling and testing the entire project once or twice per sprint.

The manager observed:

```
This comes from problems keeping my local team,
and the two remote teams we work with in In-
dia [from diverging too much].  The remote
guys tend not to commit frequently enough to
get leverage out of our continuous integra-
tion builds, even after prompting.  It is a
real challenge to know how far out of sync
[the remote teams] are [with the local team]
when their commits are not being merged in
regularly.
```

Ultimately, his desire was two-fold. First, he wanted to know when the teams were deviating so he could prompt them to merge before the situation got worse. Second, he wanted the developers to be able to identify merge problems proactively and resolve them before the they became so onerous that they were not completed until the end of the sprint.

```
I want [my developers] to at least initiate
a conversation with the relevant parties when
the system says they have, or are just or about
to, walk into a conflicting situation.  I also
want the system to give them a certain level
of trust of other developer's changes so that
if [a merge] won't cause a problem, they should
sync up.
```

His observation describes precisely the two notions of deviation we are considering. Developers can (1) underestimate how far they have deviated from others by introducing conflicts of which they are not aware, or (2) overestimate how far they have deviated from others and fail to incorporate others' changes for fear of conflicts. The research questions we address directly relate to the potential utility of a tool that could help this manager and his teams.

4. CHARACTERIZATION OF TEXTUAL CONFLICTS AND NON-CONFLICTS

Our work aims to to find opportunities for an IDE to bring previously-unavailable information to the developer's attention. Therefore, we measure how often and how early an IDE could have discovered and delivered such information to the developer. Our experiments use the histories of eight real-world open-source projects varying in size up to 1.5 million NCSL and developed by up to 100 collaborators. In this section, we define our terminology (as there is no standard terminology for version control activities), describe our subject programs, and describe the analyses and results related to textual conflicts and non-conflicts across these subjects. Our analysis is broken into three parts: how often branches and conflicts happen in these subject programs, how long unrecognized conflicting branches persist, and how long the potential for an unrecognized safe merge persists. Section 5 discusses our narrower analysis, of a single subject program, that distinguishes among different kinds of conflicts (for example, when textual merges succeed but compilation fails).

System	KNCSL	Description
Gallery3	45	Web-based photo album.
Insoshi	173	Social networking platform.
MaNGOS	520	Online game server.
Perl	599	Programming language.
Rails	118	Web application framework.
Samba	1,536	File and print services.
VLC	485	Portable multimedia player.
Voldemort	62	Structured storage system.

Figure 1: Eight subject programs we used to evaluate an IDE's access to speculative information in collaborative development environments.

4.1 Distributed Version Control Terminology

Our specific analysis relies on developer use of distributed version control systems, such as Git and Mercurial. The advantage for us is that, compared to the previously popular centralized version control systems, such as CVS and Subversion, these distributed systems keep more of a project's history. This is partly due to the capabilities of these systems and partly because of the developer habits that they enable or encourage.

The terminology is inconsistent among the different systems, so we will attempt to standardize our language for this paper. In a distributed version control system, every developer on a team has a copy of the repository on his or her local machine. At any time, the developer can perform a *local commit*, or simply a *commit*. A commit updates the status of the local repository with a snapshot of the current state of the developer's code. The snapshot itself is also called a *commit*. A developer may choose to *share* his or her code with another developer at any time. In distributed version control, sharing is typically called a `push`, and in centralized version control, a `commit`. Finally, we will use the term *update* to mean a request to get changes from another repository. In distributed version control, sharing is typically called `pull`, `update`, and/or `merge`. In centralized version control, sharing is typically known as `update`.

A repository may have any number of *branches*, each typically intended for a particular line of development. These branches may *merge*, which is essentially equivalent to a developer sharing his or her changes. Distributed repositories typically have many more branches and more frequent merging, and they capture the intermediate state of a developer's work whereas the standard use of a centralized version control system does not. At any given time, the latest commit in each branch is called a *tip*.

4.2 Subject Programs

To evaluate our approach we chose a sample set of systems that use the Git distributed version control system; these systems are summarized in Figure 1. We selected several recognizable systems from the list of projects maintained by the Git project, as well as some highly-ranked systems on

			Historical Merges				Potential Merges			
Project	# committers	# commits	total	clean	conflict	% conf.	total	clean	conflict	% conf.
Gallery3	24	5,292	563	483	80	14%	7,460	6,189	1,271	17%
Insoshi	15	1,318	114	87	27	24%	1,742	1,006	736	42%
MaNGOS	27	3,750	176	118	58	33%	4,994	3,902	1,092	22%
Perl	52	34,795	1,394	1,155	235	17%	**6,210**	**5,237**	**973**	**16%**
Rails	53	13,089	393	336	57	15%	10,970	7,922	3,048	28%
Samba	59	59,390	750	649	100	13%	**6,051**	**3,702**	**2,349**	**39%**
VLC	100	38,362	57	47	10	18%	1,815	1,672	143	8%
Voldemort	23	1,324	167	128	39	23%	4,807	3,222	1585	33%
Total	353	157,320	3,614	3,003	606	17%	44,049	32,852	11,197	25%

Figure 2: Summary of the nature of the development of eight collaborative projects. The emphasized cells indicate those part of the analysis for which we sampled only a portion of the very large project history.

GitHub.[1] We only dismissed potential systems if they had fewer than 10 developers and 1000 commits.

4.3 Conflict Frequency

The first question we asked about collaborative environments is "How often do conflicts happen?" However, in order to be able to place the answer to that question in context, we must first answer "How often do branches happen in collaborative environments?"

Figure 2 indicates the number of branches that existed at any given point in time. During virtually all of the development time, each of these projects had at least two branches. This first finding indicates that an IDE would, at the very least, have alternate branches of development which it could speculate could be merged. Further, on average, each branch contained 44 commits.

Figure 2 also shows how often merges happened in the subject projects. The number of merges ranged from 57 for VLC to 1,394 for Perl. Of these merges, 17% had textual conflicts, meaning that Git could not merge the two relevant branches itself and required human effort. This finding leaves open the possibility that an IDE could have detected these conflicts earlier than the time when the developers first attempted to merge the branches, and could have warned the developers about the conflicts while the relevant code was still fresh in their minds. We will examine this possibility in Section 4.4. The other 83% of the merges had no textual conflicts, leaving open the possibility that an IDE could have let the developers know that it was safe to incorporate others' changes at an earlier time, reducing the deviation between the branches. We will examine this possibility in Section 4.5.

Finally, we generated all possible potential merge points. These are all pairs of commits that, at some point in time, were both the tips of their respective branches. These pairs represent the potential merges a developer could expect to execute at any given time. Because of the large number of these merge points and the time necessary to attempt each merge, for some of the projects with very large histories, we were only able to analyze a subset of the potential merges, indicated in **emphasis** in Figure 2. We found that, on average, 25% of the potential merges would have resulted in a textual conflict, again possibly allowing the IDE to notify the developer early of a potential conflict and, conversely, increase a developer's confidence in the 75% of the cases that had no textual conflicts.

[1] `http://github.com`

4.4 Textually Conflicting Branch Persistence

Section 4.3 showed that branches are common. How frequent are the conflicts, and how long do they persist? This information addresses our research question "How much earlier, if at all, could an IDE discover a potential conflict than it is being discovered by engineers today?"

We examined the development history of four of our subject programs (Gallery3, Insoshi, MaNGOS, and Voldemort) to quantify the lifespan of a conflict. We considered only branches that at some point merged in the history, thus ensuring that conflicts between branches that were never intended to merge do not corrupt our results.

For each program, we performed the following analysis. First, we found all pairs of coexisting branch tips in the repository. That is, for every point in time, for every latest commit in a branch, we paired it with the latest commit in each of the other branches at that time. These pairs represent what an IDE could have observed in a collaborative environment at the time of development. Second, for each pair, we determined whether the commits could be merged using Git's built-in merging mechanism and whether that merge resulted in a conflict. Whenever we detected a conflict, we found the commit in the repository that resulted in a persistent merge of those branches. Sometimes, these merges happened immediately after the introduction of a conflict while other times, they merged almost a year later with hundreds of new commits occurring during that time. Finally, we ignored conflicts between a pair of tips that was already evidenced by an earlier pair. That is, we did not double count the conflicts; if two branches conflicted, we counted only the longest life of the conflict, and not the conflicts of the pairs that may occur prior to the actual merge.

Figure 3 shows the conflict persistence data for our four subject programs. In these four programs, 20% of the branch pairs that did eventually merge conflicted textually. These conflicts, on average, persisted for 9.8 days and developers made, on average, 11.6 commits, on each branch, before merging. Thus, a speculative IDE with access to the collaborative environment could, on average, have let developers know 9.8 days and 11.6 commits earlier about an existing conflict. In the worst case, one conflict in MaNGOS persisted for 334 days and included 676 commits along one of its branches.

The 9.8 day average, although significant, is muddied by some limitations of our evaluation approach. Specifically, the repository can tell us when a conflict first arose and

Project	Merges	Conflicting Branches	% Conflicting	Conflicting Branches time mean	time max	# commits mean	# commits max
Gallery3	563	80	14%	3.1 days	57 days	7.5	229
Insoshi	114	27	24%	11.7 days	101 days	9.4	81
MaNGOS	176	58	33%	8.2 days	334 days	17.6	676
Voldemort	167	39	23%	25.7 days	147 days	12.8	89
Total	1020	204	20%	9.8 days	334 days	11.6	676

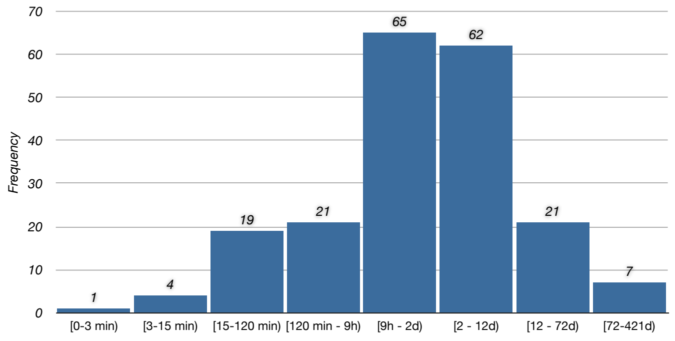

Figure 3: Conflict Persistence.

when the developers resolved it. However, the repository does not tell us (1) when the developers may have found out about the the conflict, (2) if the developers began resolving the conflict as soon as learning of its existence, and (3) had the developers known about the conflict earlier, would they have done anything differently to resolve it. Such information is unavailable from version control repositories. However, these shortcomings do not alter the fact that the information about the conflict was available earlier, and that the IDE could have brought it to the developers' attention at that time, allowing the developers to make a better-informed decision about how to proceed.

Because textual conflicts are a subset of all possible conflicts, our analysis is conservative. Two branches may merge cleanly from Git's textual perspective, but the merged result may fail to compile, fail to pass a test suite, or fail along some other analysis dimension. Thus, the numbers we have provided in this section are an underestimate of the times the IDE can provide conflict information to the developer. (Again, Section 5 discusses other types of conflicts, how the IDE can leverage other types of analysis to bring the developer more information on these conflicts, and how incorporating this analysis affects our estimates.)

The lifespan of a conflict addresses one of our research questions: whether an IDE could provide engineers with information about potential conflicts early in the development process. This question is aimed toward one of the two divergence phenomena we described: engineers mistakenly believing that they have diverged less from their collaborators than they in fact have. Bringing the existence of conflicts to their attention may help alleviate this misconception.

4.5 Textually Safe Branch Persistence

The lifespan of a conflict only covers one dimension of the problem. Here we describe what information is available to the IDE with regard to the second phenomenon: engineers mistakenly believing that they have diverged more from their collaborators than they in fact have.

At such times, an engineer may not incorporate a collaborator's changes for fear that such incorporation could break the engineer's code. Unnecessarily keeping one's code out of date can cause greater deviations and more-severe conflicts than if the safe changes are incorporated early and often. The logical counterpart to our analysis of an IDE's ability to detect conflicts is the analysis of the ability to detect safe merges so that the IDE could inform the developer that it is safe to incorporate others' changes.

Again, we examined the development history of four of our subject programs, Gallery3, Insoshi, MaNGOS, and Voldemort, and again, we ignored branches that, to date, have not merged because the developers' intent for those branches is unclear. Just as before, for each program, we found all pairs of coexisting branch tips in the repository. We then, for each pair, determined whether the commits could be merged using Git's built-in merging mechanism and whether that merge resulted in a conflict. Whenever Git's mechanism succeeded, we knew that the branches could have merged textually cleanly. We then found the time in the repository when the branches actually merged. As before, we ignored pairs of tips that were already evidenced by an earlier pair and thus did not double count; if two branches merged textually cleanly, we counted only the longest life of the clean merge, and not the possible clean merges that may occur prior to the actual merge.

We found that, in these four programs, 80% of the branch pairs (the complement to the 20% conflicting branch pairs) that did eventually merge had no textual conflicts. These branches, on average, persisted for 11 days and developers made, on average, 11.5 commits on each branch before merging. Thus, on average, developers using the speculative IDE could have known about safe merging and could possibly have stayed up-to-date 11 days and 11.5 commits earlier. In the worst case, one branch pair in Voldemort persisted for 138 days and another in Gallery3 persisted for 232 commits without a merge, while any and all of the number of possible merges along the way would have been textually clean.

Project	Merges	Non-Conflicting Branches	% Non-Conflicting	Non-Conflicting Branches time mean	time max	# commits mean	# commits max
Gallery3	563	483	86%	.65 days	7.4 days	14.3	232
Insoshi	114	87	76%	3.4 days	39 days	6.3	64
MaNGOS	176	118	67%	2.4 days	10 days	5.8	44
Voldemort	167	128	77%	35 days	138 days	9.7	77
Total	1020	816	80%	11 days	138 days	11.5	232

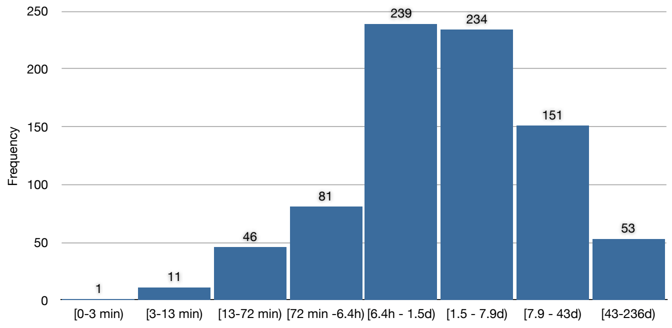

Figure 4: Textually Safe Merge Persistence.

Figure 4 shows the persistence of textually safe branches for our four subject programs. Again, the log-scale histogram demonstrates the approximately power-law-like distribution of the conflict lifespan.

Just as with conflict detection, learning information about textually safe merges from repositories has its limitations. The 11-day improvement in mean time to incorporating others' changes is significant, but the repository does not tell us (1) when the developers knew merging was safe, (2) if the branches merged as soon as the developers knew it was safe, and (3) whether knowing about the merging safety would have influenced the developers to act differently. Nevertheless, the fact remains that the information about the textually safe merge was available earlier, and that the IDE could have brought it to the developers' attention at that time, allowing the developers to make a well-informed decision about how to proceed.

In contrast to the textual conflict analysis, our analysis here is not conservative. That is, we found that 80% of branches did not conflict textually, whereas they may have had higher-level conflicts related to compilation, testing, and other analysis. In Section 5, we will discuss the other types of conflicts that can take place and how incorporating this analysis affects our estimates. However, it should be noted that whatever error is present in our estimates of this section, it directly benefits the estimates of Section 4.4.

5. HIGH-ORDER CONFLICTS

The most common and familiar kind of conflict identified in version control is a textual conflict. When developer Melinda shares her code with developer Bill, the version control system will examine Melinda's and Bill's changes since their last share. Even if they both made changes, but those changes are in independent files, or even distant parts of the same files, the merge of the two branches will execute smoothly. However, if Melinda and Bill changed the same, or proximate lines of the same files, the version control system will declare a a conflict and ask either Melinda, or Bill, or both, to resolve the conflict by hand.

However, textual conflicts are only one kind of a conflict. Even though branches may merge without conflicting textually, the resulting code may fail to compile, fail to pass a unit test suite, fail a set of system tests, fail in some qualities of service, or fail along some other dimension of analysis. When an IDE speculates merges and presents the developer with information on that merge, it could present not only textual conflicts but also these higher-level conflicts. Access to this information would allow the developer to make better informed decisions about the affects of sharing.

To determine whether these high-order conflicts occur frequently enough to warrant the IDE attempting to observe them, we further analyzed the Voldemort system. We first determined how many of the branches that were textually safe were not safe from the points of view of three high-order analyses: compilation, halting of the test suite, and behavior. We then, examined a merge that failed the behavioral analysis and a merge that failed the compilation analysis, as two case studies of impact of such analysis.

We now describe the three types of high-order analysis we performed. **Compilation** analysis attempted to compile the speculated merges of two branches; of the 128 branch pairs that were textually safe, 12 (9.4%) failed to compile. **Halting** analysis executed a set of tests over those speculated merges that compiled successfully; of the 116 branch pairs, 4 (3.4%) entered infinite loops. **Behavioral** analysis executed a test suite over those speculated merges that did not enter infinite loops to examine the correctness of the program's behavior; of the 112 branch pairs, 8 (7.1%) failed at least one test that the tips of the branches used to speculate the merge passed.

Figure 5 illustrates our high-order analysis results. Overall, of the 167 branch points in Voldemort, 104 (62%) contained no conflicts. For the other 63 (38%), the IDE could have reported information on a conflict to the developer when it first became available.

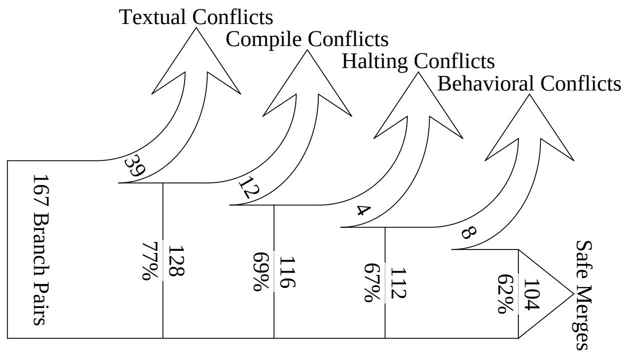

Figure 5: High-order conflict analysis revealed that 14% of all branch pairs and 19% of those branch pairs that were textually safe were deemed unsafe by one of our high-order analyses.

5.1 Malformed Non-Code Resource

On October 10, a developer successfully merged two tips (`50b74` and `00c35`). Tip `00c35` was edited 17 times while the branch was alive and the last commit on this branch occurred only 8 minutes before the merge. Tip `50b74` had not been edited in the previous 48 days. Although the patch between these two tips was very large (63,413 lines), the merged system successfully executed its test suite. However, the merged system failed 15 tests: while 14 of them failed in both `50b74` and `00c35`, one test, (`voldemort.store.http.HttpStoreTest::testBadPort()`) did not fail either tip before the merge, but did in the merged system. Thus some unintended behavioral interaction between the two branches' changes broke this test. Further investigation revealed that the merge invalidated one of the metadata files, `cluster.xml`. In this case, if the IDE had let the developers know that it was safe to merge earlier, the problem could have been avoided completely.

5.2 Build Breaks After Merge

On November 9, a developer successfully merged tips `c77a4` and `7f776`. Tip `7f776` was edited 11 times while the branch was alive; tip `c77a4` was edited 3 times. Both tips had been modified within 4 days of the merge. While the merge had no textual conflicts, the code failed to compile: 4 compilation errors resulted from referencing a missing type `ProtoBuffAdminClientRequestFormat`. Eight minutes after this merge, the developer merged in another tip (`68e3b`), which resolved the compilation problem.

In this case, the IDE could have speculatively told the developer about the compilation error that would arise after the merge. With this information, the developer may have chosen to do the merges in an alternate order to avoid the problem and ensure other developers were not adversely affected.

6. RELATED WORK

In this section, we discuss the states-of-the-art in version control systems, collaborative awareness, mining software repositories, and continuous development, and compare and contrast them to this paper's contributions.

6.1 Version Control Systems

Rochkind introduced the first source code control system in 1975 [19]. Since then, numerous similar systems — characterized by a centralized shared repository — have been developed and deployed, among them RCS [26], CVS [8], Subversion [4], and others. More recently, a set of distributed version control systems have been developed including Bazaar, Mercurial, and Git.[2] These systems do not rely on a centralized repository, allow more freedom to the collaborators in terms of branching, merging, and keeping multiple repositories, and are less dependent on network availability.

The distinctions between the conventional and distributed version control approaches are significant, as are the distinctions among specific version control systems [5, 14, 16]. These distinctions do not, however, affect our research in fundamental dimensions. The only exception to this might be that distributed version control systems encourage more frequent branching and merging, which likely provides additional opportunities for speculation [27]. In any case, Perry et al. [17] empirically document the variations, and consequences of the variations with respect to quality and schedule, in how software teams perform work in parallel.

6.2 Collaborative Awareness

There is significant related literature that addresses, in varying dimensions, the importance of increasing awareness of the activities among members of collaborative software teams. Earlier studies documented the use of existing tools such as mailing lists and chat systems to increase awareness among team members [9]. FASTDash [3] is an example of an interactive visualization designed to augment existing software development tools with a specific focus on helping people understand what other team members are doing; they produce a spatial representation of the shared code base to highlight these activities. Dewan and Hegde [6] reported on the CollabVS can smooth collaborative version control by identifying conflicts by analyzing dependencies among program elements in checked-out versions.

A collection of results by Sarma and colleagues include tools that support cooperative software development tasks, with a primary focus on aspects of social dependencies [24, 23, 1]. Their Palantír work, in particular, has similar motivations to ours: "[P]roviding workspace awareness to users will enable them to detect potential conflicts earlier, as they occur. Ideally developers can then proactively coordinate their actions to avoid those conflicts" [24, p. 1]. Palantír contrasts with our approach by showing which developers are changing which artifacts by how much. Our approach, on the other hand, considers not only identifying conflicts earlier but also considers when updates can take place safely earlier; also, we consider multiple levels of conflicts — textual, syntactic, and behavioral.

Sarma, Redmiles, and van der Hoek [25] not only empirically assess the benefits of collaborative awareness for configuration management, but they also compare their assessment to those provided by FASTDash [3] and CollabVS [6]. In contrast to these three studies, which use observational and laboratory studies, our approach is analytic in assessing the potential for benefits. In addition, we look not only for conflicts but also for missed safe merging opportunities. That said, our analysis is consistent with these studies in

[2]`http://bazaar.canonical.com`, `http://mercurial.selenic.com`, and `http://git-scm.com`, respectively.

confirming the potential for better coordination of individual and team repositories.

6.3 Mining Software Repositories

Extracting data from existing software repositories as a way to learn about more effective methods to develop software has become increasingly common over the past decade. In general, research in this area identifies one or more general questions and then assess one or more repositories to find information material to those questions, often seeking correlations among various metrics.

An early effort by Ball et al. [2] extracted metrics such as coupling — based on the probability that two classes are modified together — and used the metrics to assess the relationship between implementation decisions and the evolution of the resulting system. A number of later efforts mine version histories to determine functions that must likely be modified as a group [28], to identify common error patterns [13], to predict component failures [15], etc.

Our effort contrasts, to a large degree, with these efforts in at least two dimensions. First, the property we are assessing — unexploited and promising opportunities to incorporate or share changes with others on a team — appears to be distinct. Second, we are mining only to determine if this property occurs frequently enough to justify the development of a supporting mechanism. Other mining efforts tend to look for patterns that might inform more general software development improvements (for instance, allocating more quality assurance resources to more error-prone components).

6.4 Continuous Development

Our approach can be characterized, or perhaps more accurately inspired by, the notion of continuous merging. Thus, it is related to a number of other approaches to continuous computation in the context of software development.

Over two decades ago Henderson and Weiser [10] proposed a programming environment modeled on spreadsheets, in which the program being developed was continuously executed as it was being edited; they explicitly anticipated powerful personal computers as requirements for effective implementation. They pursued dimensions of VisiProg for several years [12], but for nearly two decades afterward the primary focus in this area — to a limited degree, at least — was on incremental computation for general program manipulation [18, 11] without a concentration on environments, *per se*.

Modern programming environments provide continuous compilation. The environment maintains the project in a compiled state as it is edited, speeding software development in two ways. First, the developer receives rapid feedback about compilation errors on every save, allowing for quick correction while that code is fresh in the developer's mind. Secondly, the developer is freed from deciding when to compile, meaning that when it is time to run or test the code, no intervening compilation step is necessary.

Continuous testing [20, 21, 22, 7] uses excess cycles on a developer's workstation to continuously run regression tests in the background. It is intended to reduce the time and energy required to keep code well-tested and prevent regression errors from persisting uncaught for long periods of time. The vision is that after every keystroke, the developer knows immediately (without taking any extra action) whether the change has broken the tests. Continuous testing has gained some traction in the development community.

These continuous approaches are *reactive*, albeit very fast. In contrast, our notion of speculative version control relies instead on computing (and perhaps presenting, depending on the user interface and user preferences) contingent information about version control operations *before* the programmer has even considered taking the associated speculative action.

7. CONTRIBUTIONS

In today's distributed collaborative environment for software development, it is easy for members of engineering teams to become unaware of how far they may have diverged from their teammates. In this paper, we set out to learn whether adequate opportunities exist for an IDE to provide pertinent information to developers to (1) inform them of potential conflicts early, and (2) increase their confidence when their work does not conflict with that of others. By studying eight real-world projects, each with up to 1.5 million lines of code and up to 100 developers, we found that there are ample opportunities for an IDE to deliver such information to the developers. In particular, branches are abundant in software development and 17% of these branches contain textual conflicts that persist for an average of 10 days before they are resolved. Further, non-conflicting branches persist for an average of 11 days without developers sharing their work. Finally, higher-order analysis revealed that another 19% of the branches do not conflict textually but do conflict syntactically or behaviorally. These findings justify the creation of a mechanism within an IDE to bring speculative merge information to the attention of the developer.

Perhaps primarily because we were acutely attuned to these issues, during the collaborative development of the software infrastructure for the experiments described earlier, we repeatedly encountered times when we wanted to synchronize our view of the source code with that of the others but usually did not do so because we were scared that such synchronization would break our view. While advance conflict warning may perhaps prevent costlier problems, in our recent experience and consistent with our analysis, safe merge detection would likely be used more often by the developers to increase their confidence and comfort level with respect to how far they are diverging from the rest of their team.

8. REFERENCES

[1] B. Al-Ani, E. Trainer, R. Ripley, A. Sarma, A. van der Hoek, and D. Redmiles. Continuous coordination within the context of cooperative and human aspects of software engineering. In *CHASE '08: Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering*, pages 1–4, New York, NY, USA, 2008. ACM.

[2] T. Ball, J.-M. Kim, A. A. Porter, and H. P. Siy. If your version control system could talk .... In *ICSE '97 Workshop on Process Modelling and Empirical Studies of Software Engineering*, 1997.

[3] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson. Fastdash: a visual dashboard for fostering awareness in software teams. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1313–1322, New York, NY, USA, 2007. ACM.

[4] B. Collins-Sussman. The subversion project: buiding a better cvs. *Linux J.*, 2002(94):3, 2002.

[5] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Comput. Surv.*, 30(2):232–282, 1998.

[6] P. Dewan and R. Hegde. Semi-synchronous conflict detection and resolution in asynchronous software development. In L. Bannon, I. Wagner, C. Gutwin, R. Harper, and K. Schmidt, editors, *ECSCW'07: Proceedings of the Tenth European Conference on Computer Supported Cooperative Work*, pages 159–178. Springer, September 2007.

[7] D. S. Glasser. Test factoring with `amock`: Generating readable unit tests from system tests. Master's thesis, MIT Dept. of EECS, Aug. 21, 2007.

[8] D. Grune. Concurrent versions systems, a method for independent cooperation. Technical Report IR 113, Vrije Universiteit, 1986.

[9] C. Gutwin, R. Penner, and K. Schneider. Group awareness in distributed software development. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 72–81, New York, NY, USA, 2004. ACM.

[10] P. Henderson and M. Weiser. Continuous execution: The VisiProg environment. In *ICSE*, pages 68–74, Aug. 1985.

[11] G. E. Kaiser. Incremental dynamic semantics for language-based programming environments. *ACM Trans. Program. Lang. Syst.*, 11(2):169–193, 1989.

[12] R. R. Karinthi and M. Weiser. Incremental re-execution of programs. In *SIGPLAN '87: Papers of the Symposium on Interpreters and interpretive techniques*, pages 38–44, New York, NY, USA, 1987. ACM.

[13] B. Livshits and T. Zimmermann. DynaMine: Finding common error patterns by mining software revision histories. In *ESEC/FSE*, pages 296–305, Sep. 2005.

[14] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28(5):449–462, 2002.

[15] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *ICSE*, pages 452–461, May 2006.

[16] B. O'Sullivan. Making sense of revision-control systems. *Queue*, 7(7):30–40, 2009.

[17] D. E. Perry, H. P. Siy, and L. G. Votta. Parallel changes in large-scale software development: an observational case study. *ACM Trans. Softw. Eng. Methodol.*, 10(3):308–337, 2001.

[18] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 502–510, New York, NY, USA, 1993. ACM.

[19] M. J. Rochkind. Mining metrics to predict component failures. *IEEE TSE*, 1(4):364–370, 1975.

[20] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *ISSRE*, pages 281–292, Nov. 2003.

[21] D. Saff and M. D. Ernst. Continuous testing in Eclipse. In *2nd Eclipse Technology Exchange Workshop (eTX)*, Barcelona, Spain, Mar. 2004.

[22] D. Saff and M. D. Ernst. An experimental evaluation of continuous testing during development. In *ISSTA*, pages 76–85, July 2004.

[23] A. Sarma, G. Bortis, and A. van der Hoek. Towards supporting awareness of indirect conflicts across software configuration management workspaces. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 94–103, New York, NY, USA, 2007. ACM.

[24] A. Sarma, Z. Noroozi, and A. van der Hoek. Palantír: raising awareness among configuration management workspaces. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 444–454, Washington, DC, USA, 2003. IEEE Computer Society.

[25] A. Sarma, D. Redmiles, and A. van der Hoek. Empirical evidence of the benefits of workspace awareness in software configuration management. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 113–123, New York, NY, USA, 2008. ACM.

[26] W. F. Tichy and W. F. Tichy. Rcs - a system for version control. *Software - Practice and Experience*, 15:637–654, 1985.

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

[28] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE*, pages 563–572, May 2004.