Proactive Detection of Collaboration Conflicts

Yuriy Brun[†], Reid Holmes[♣], Michael D. Ernst[†], David Notkin[†]
[†]Computer Science & Engineering
University of Washington
Seattle, WA, USA
[♣]School of Computer Science
University of Waterloo
Waterloo, ON, Canada
{brun,mernst,notkin}@cs.washington.edu, rtholmes@cs.uwaterloo.ca

Abstract

Collaborative development can be hampered when conflicts arise because developers have inconsistent copies of a shared project. We present an approach to help developers identify and resolve conflicts early, before those conflicts become severe and before relevant changes fade away in the developers' memories. This paper presents three results.

First, a study of open-source systems establishes that conflicts are frequent, persistent, and appear not only as overlapping textual edits but also as subsequent build and test failures. The study spans nine open-source systems totaling 3.4 million lines of code; our conflict data is derived from 550,000 development versions of the systems.

Second, using previously-unexploited information, we precisely diagnose important classes of conflicts using the novel technique of speculative analysis over version control operations.

Third, we describe the design of Crystal, a publicly-available tool that uses speculative analysis to make concrete advice unobtrusively available to developers, helping them identify, manage, and prevent conflicts.

Categories and Subject Descriptors: D.2.3 [Software Engineering]: Coding Tools and Techniques; D.2.6 [Software Engineering]: Programming Environments; D.2.8 [Software Engineering]: Metrics: process metrics
General Terms: Design, Human Factors, Experimentation
Keywords: collaborative development, collaboration conflicts, developer awareness, speculative analysis, version control, Crystal

1. Introduction

Each member of a collaborative development project works on an individual copy of the project files (source code, build files, etc.). Each developer repeatedly makes changes to his or her local copy of the files, shares those changes with the team, and incorporates changes from teammates.

The loose synchronization of these activities permits rapid development progress, but also allows two developers to make simultaneous, conflicting changes. Such *conflicts* [8, 11, 13, 17, 23, 34] are costly: they delay the project while the conflict is understood and resolved. Fear of conflicts is also costly. A developer may choose to postpone the incorporation of teammates' work because of a concern that a conflict may be hard to resolve [8, 13]. Ironically, this fear of *potential* conflicts can cause developer copies to diverge, making *real* conflicts more likely.

Conflicts can be *textual* or *higher-order*. A textual conflict arises when two developers make inconsistent changes to the same part of the source code. To prevent subsequent changes from overwriting previous ones, a version control system (VCS) allows the first developer to publish changes, but precludes the second developer from publishing until the conflict is resolved automatically (by the VCS) or manually (by a developer). Higher-order, and likely more damaging conflicts arise when the VCS can integrate the developers' textual changes, but the changes are semantically incompatible and can cause compilation errors, test failures, or other problems. Such conflicts are problematic to detect and resolve in practice [17].

As with errors in programs, it is generally easier and cheaper to identify and fix conflicts early, before they propagate in the code and the relevant changes fade away in the memories of the developers. Currently, this information is not readily available to developers [9].

Our approach, speculative analysis, unobtrusively provides information about the presence or absence of conflicts in a continual and accurate way. Our intent is for this information to allow developers make better-informed decisions about how and when to share changes, while simultaneously reducing the need for human processing and reasoning. We make the following contributions:

- We analyze nine open-source systems and show that, in practice, conflicts between developers' copies of a project are: (1) the norm, rather than the exception, (2) persist, on average, 10 days, and (3) are often higher-order.
- Using previously-unexploited information, we precisely diagnose important classes of conflicts and offer concrete advice about addressing them. We do this by introducing a novel technique called speculative analysis that anticipates actions a developer may wish to perform and executes them in the background. Reporting the consequences of these likely version control operations can improve the way in which collaborative developers identify and manage conflicts.
- We design and implement an open-source, publicly-available tool called Crystal[1] — `http://crystalvc.googlecode.com` — that implements the analyses and unobtrusively presents advice to developers, to aid them in identifying, managing, and preventing conflicts.

Section 2 provides a brief scenario of two collaborative developers, sketching how their development activities would differ with

[1] Crystal "exceeded expectations" of the ESEC/FSE 2011 artifact evaluation committee.

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

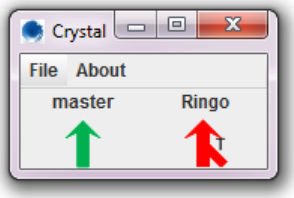

Figure 1: A screenshot of the Crystal tool as run by a developer named George. The green arrow informs George that his changes can be published (uploaded) without conflict to the master repository. The red merge symbol indicates that Ringo's changes, if combined with George's, would cause a test ("T") failure.

and without the use of a Crystal-like tool. Section 3 presents VCS terminology. Section 4 details our retrospective analysis of the frequency and persistence of conflicts in practice. Section 5 describes the information that can help developers better manage their conflicts. Section 6 introduces the design of Crystal, an unobtrusive tool that computes and reports this information to developers. Section 7 surveys related work. Section 8 discusses threats to validity. Finally, Section 9 summarizes our results and contributions.

2. Scenario

Consider a simple scenario with George and Ringo adding features to a project. As part of George's feature, he makes changes to a library that Ringo also uses. When finished, George and Ringo each independently publish their changes to the master repository. Whenever the regression tests run after both have published, George and Ringo are notified about failing tests. At that time, they have to recollect their earlier changes and assumptions, and their fixes might force them to rework other code they had written in the meanwhile.

One way to lessen these difficulties is to use an awareness tool, which reports where in the code base teammates are working, allowing a developer to be more attentive to conflicts that may arise in those locations (see Section 7 for more details). For example, when George edits the library, an awareness tool may tell Ringo that someone else is editing code he depends on. However, if George's change to the library had not actually affected Ringo, the warning would have been a false positive. Furthermore, George might have been exploring some ideas and changes, without ever intended to share the intermediate changes with his team. Thus, awareness tools have the potential to give early warnings, but also the potential to give multiple types of false warnings.

By contrast, consider what would have happened if George and Ringo were using a speculative analysis tool, such as our tool Crystal, that proactively informs developers of version control conflicts. Figure 1 shows how Crystal informs George that his library changes' integration with Ringo's changes causes Ringo's test suite to fail. The tool encourages George and Ringo to address the impending conflict before their assumptions and understanding of the changes fade.

Speculative analysis [4] does not guess at possible conflicts. Instead, it speculatively performs the work and executes the VCS operations in the background on clones of the program: it actually merges George's and Ringo's committed code, builds it, and runs its tests. This allows speculative analysis to deliver precise information about conflicts: those that can be merged safely are not reported as potential conflicts, and textually-clean merges that fail to build or test properly are reported as conflicts. With respect to the issue of exploratory development and awareness tools, our approach assumes that when a developer commits code to the VCS, the developer decided to share that code with other developers. Overall, our approach provides precise and pertinent information available as soon as conflicts occur in the VCS.

3. Terminology

Our results are applicable in the context of both centralized version control systems (CVCSes) — such as CVS, Subversion, and Perforce — and distributed version control systems (DVCSes) — such as Git and Mercurial. This paper focuses on DVCSes to simplify the presentation. We first briefly present accepted DVCS terminology. We then introduce additional new terminology to allow us to precisely characterize seven pertinent relationships between repositories.

3.1 Version control terminology

Figure 2 shows a common [31] DVCS repository setup. There is a single *master* repository and four developers: George, Paul, Ringo, and John. Each developer makes a local repository *clone* from the master. Each local repository contains a complete and independent history of the master repository at the time it was cloned. In addition, each repository has a *working copy*, in which code is edited. Changing the working copy does not modify the local repository; to modify the local repository the developer *commits* a *changeset* to the local repository's history. Teammates are not privy to these changesets until the developer *pushes* them to the master repository from the local repository. After a push, another developer can perform a *pull* from the master, which updates that developer's local repository with the changesets. To refresh a working copy after a pull, the developer must apply the *update* operation. It is common for developers in a DVCS to commit multiple times before publishing through a push. It is uncommon for developers to pull without a corresponding update. A *merge* conflict can arise due to a pull operation, and the conflict must be *resolved* before proceeding. The terms used above are common, or have direct equivalents, across DVCS systems.

The discussion in this paper makes a few simplifying assumptions for clarity: one, it assumes that developers push to and pull from only the master repository; two, it assumes that developers only make a commit when all their tests pass. However, our approach in general, and the Crystal tool, in particular, handle arbitrary pushes, pulls, and commits.

3.2 Repository relationships

We have identified seven relevant relationships that can hold between two repositories. Figure 2 illustrates these relationships.

SAME**:** The repositories have the same changesets. For example, George's repository is the SAME as the master repository because they both consist of changesets 100 and 101.

AHEAD**:** The repository has a superset of the other repository's changesets. For example, George's repository is AHEAD of Paul's.

BEHIND**:** The inverse of AHEAD; for example, George's repository is BEHIND John's.

The remaining four relationships represent repositories that share an initial sequence of changesets followed by distinct sequences of changesets.

TEXTUAL✘**:** (pronounced "textual conflict") The distinct changesets necessitate human intervention as they cannot be automatically merged by the VCS. For example, if George's changeset 101 and Ringo's changeset 102 modify overlapping lines of code, they are in TEXTUAL✘.

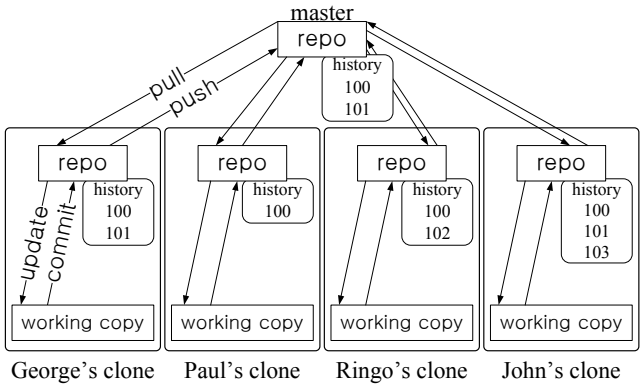

Figure 2: A DVCS with four clones of a master repository. The box labeled "history" lists those changesets currently in the repository. Each repository may have a working copy. The commit command creates a new changeset in its repository's history, and the update command incorporates changesets into the working copy. A developer can incorporate changesets from other repositories using the pull command and can share changesets using the push command.

BUILD✘: The repositories can be automatically merged by the VCS, but the resulting merged code fails to build.

TEST✘: The repositories can be automatically merged by the VCS and the resulting merged code builds but fails its test suite.

TEST✓: The repositories can be automatically merged by the VCS and the resulting merged code builds and passes its test suite.

Analogously to TEST✓, there there are relationships BUILD✓ and TEXTUAL✓ with the obvious meanings. The table header of Figure 4 illustrates the interrelation among the relationships. When build scripts and test suites are not available, we will distinguish only five relationships: SAME, AHEAD, BEHIND, TEXTUAL✓, and TEXTUAL✘.

Higher-order conflicts, such as BUILD✘ and TEST✘, are not considered by existing VCS systems. Although this paper discusses only these two higher-order relationships, others naturally arise for other analyses; for example, consider when a test suite passes but a performance analysis or code style checker does not.

4. Conflicts in practice

This section answers the following research questions. "How often do the TEXTUAL✘, BUILD✘, TEST✘, and TEST✓ relationships of Section 3.2 happen?" Section 4.1 focuses on the TEXTUAL✘ relationship, and Section 4.2 addresses BUILD✘, TEST✘, and TEST✓. Section 4.3 asks "How long do developers experience the conflict relationships such as TEXTUAL✘?" Section 4.4 asks "How damaging is it not to share changes with teammates, if those changes would currently merge cleanly?"

Anecdotally, conflicts are a serious problem. For example, in a private communication, an industrial manager expressed the following concerns to us about his two offshore teams and their collaboration with his local team:

> "The remote guys tend not to commit frequently enough to get leverage out of our continuous integration 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.
> ...
> I want [my developers] to at least initiate a conversation with the relevant parties when the system says they have, or are just 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."

system	KNCSL	devs	changesets	days	description
Gallery3	57	24	4,838	437	Web-based photo album
Git	267	27	20,785	1,741	Version control system
Insoshi	173	15	1,316	629	Social networking platform
jQuery	26	23	2,183	1,393	JavaScript library
MaNGOS	643	27	3,511	626	Online game server
Perl5	660	51	34,653	8,061	Programming language
Rails	141	50	12,342	1,875	Web application framework
Samba	1,363	59	58,802	5,001	File and print services
Voldemort	103	22	1,219	375	Structured storage system
Total	3,433	298	138,549	20,138	

Figure 3: Nine subject programs analyzed to address RQ1, RQ2, and RQ3 in collaborative development environments. KNCSL stands for thousands of non-comment source lines. The version control history ends on Feb 13, 2010.

There is little hard data on conflicts. Zimmermann's analysis of CVS repositories for four open source systems is the only work we could find that directly addresses this issue [34]. He reported that of all merges, 23% to 47% had textual conflicts (TEXTUAL✘) while the remainder could be merged automatically (TEXTUAL✓). Answering RQ1 and RQ2 requires analyses that significantly augment these data and anecdotes.

As subject programs (Figure 3), we chose Git itself and the eight most active projects on GitHub (`http://github.com`) that satisfy the following three criteria: (1) at least 10 developers, (2) at least 1000 changesets, and (3) not just a Git copy of a CVC repository (which would not contain sufficient information to answer our research questions).

4.1 Textual conflicts

RQ1: How frequently do conflicts — textual and higher-order — arise across developers' copies of a project?

The answer to RQ1 is that conflicts are the norm: for each subject system, there were no times when all pairs of developers were in consistent relationships (SAME, AHEAD, or BEHIND) with each other.

Figure 4 shows how often developers merged their changes. (This is analogous to Zimmermann's result described above.) Of all the merges, one in six, or 17%, had textual conflicts as determined by Git's built-in merging mechanism, reflecting the TEXTUAL✘ relationship. (This number may be smaller than Zimmermann's 23–47% due to better merging algorithms in DVCSes.) The other 83% of the merges had no textual conflicts, meaning the relevant developers were in the TEXTUAL✓ (including BUILD✘ and TEST✘) relationship.

The importance of the frequency of the TEXTUAL✘ relationship is clear: an unrecognized TEXTUAL✘ between the repositories of two developers may cause problems. The importance of the frequency of the TEXTUAL✓ relationship is also material: a developer who is unsure whether others' changes can be incorporated safely might avoid doing so, allowing conflicts to persist and grow (as suggested in the manager's quotation above).

Figure 5 considers every commit at which developers who *did* eventually merge their changes *could* have done so earlier. On average, 19% of the potential merges would have resulted in a textual conflict. In other words, had the developers been using Crys-

system	merges	TEXTUAL✘		TEXTUAL✓					
				BUILD✘		BUILD✓			
						TEST✘		TEST✓	
Git	1,362	227	17%	2	.1%	53	4%	1,080	79%
Perl5	185	14	8%	7	4%	51	28%	113	61%
Voldemort	147	25	17%	15	10%	5	3%	102	69%
Gallery3	458	42	9%	416		91%			
Insoshi	93	23	25%	70		75%			
jQuery	15	1	7%	14		93%			
MaNGOS	192	81	42%	111		58%			
Rails	362	51	14%	311		86%			
Samba	748	100	13%	648		87%			
total	3,562	564	16%	2,998		84%			

Figure 4: Historical merges. Frequencies with which developers experienced TEXTUAL✘, BUILD✘, TEST✘, and TEST✓ relationships when they integrated their code. For three systems with non-trivial test suites in the repository, we measured the frequencies of all four relationships; for the other six (which had no non-trivial test suite that we could run), we measured only TEXTUAL✘ and TEXTUAL✓.

tal, it would have informed developers about TEXTUAL✘ relationships that resulted from 19% of the commits. Conversely, the 81% of clean merges indicate the likely benefit of notifying developers when a safe textual merge can be performed.

4.2 Higher-order conflicts

In our subject programs, 17% of merge operations required human assistance to resolve a textual conflict (Figure 4). This underestimates the human effort, since textually-safe merges are not always safe: an automatically merged change may suffer a build or test failure, for example. We computed the relationships at the time of each of the 5,355 merges that developers performed during the development of Git, Perl5, and Voldemort. We did not compute the information for the other six subject program because of the absence of a non-trivial test suite that we could run.

Figure 4 show that during the development of Git, Perl5, and Voldemort, 76% of merges completed cleanly, 16% of merges resulted in a textual conflict (TEXTUAL✘), 1% of merges resulted in a build failure (BUILD✘), and 6% of merges resulted in a test failure (TEST✘). The 266 textual conflicts reported by the version control system only represent 67% of all conflicts. Stated another way, 33% of the 399 merges that the version control system reported as being a clean merge, actually were a build or test conflict.

Few current awareness tools detect higher-order conflicts (see Section 7). Rather, they generally notify developers of all changes to the repository (e.g., FASTDash [3]) or of concurrent changes to ASTs (e.g., Syde [15]). In contrast, we use the project's tool chain to dynamically and precisely detect BUILD✘ relationships (via the build system) and TEST✘ relationships (via the test suite).

4.3 Persistence of conflicts

RQ2: How long do conflicts persist?

RQ2 asks how long developers experience the TEXTUAL✘ relationship. As we argue in Section 4.4, the longer a relationship persists, the more opportunities it has to change into a more severe relationship.

To measure the lifespan of a conflict, for each conflict that was eventually resolved, we found the first changeset that introduced the conflict and the changeset that resolved it. (We omitted all conflicts between changesets that were never actually merged in the history, such as those on dead-end branches.) Due to the sheer volume of the data and computation necessary to process it, we examined the development histories of only four of our subject programs: Gallery3, Insoshi, MaNGOS, and Voldemort.

system	merges	TEXTUAL✘		TEXTUAL✓	
Git	179,249	15,965	9%	163,284	91%
Perl5	7,352	1,290	18%	6,052	82%
Voldemort	4,512	1,534	34%	2,978	66%
Gallery3	6,924	1,262	18%	5,662	82%
Insoshi	1,742	736	42%	1,006	58%
jQuery	74	13	18%	61	82%
MaNGOS	4,967	1,092	22%	3,875	78%
Rails	10,418	2,971	29%	7,447	71%
Samba	77,683	30,635	39%	47,048	61%
total	292,921	55,498	19%	237,423	81%

Figure 5: Potential early merges. The frequency with which developers would be informed of TEXTUAL✘ and TEXTUAL✓ relationships, if they had used Crystal throughout their development of nine open-source systems.

On average, the TEXTUAL✘ relationship persisted for 9.8 days and involved 23.2 changesets — 11.6 per developer — (with median values of 1.6 days and 3 changesets) before incorporating (left side of Figure 6). A tool could have let developers know about these TEXTUAL✘ relationships immediately upon their creation. In the worst case, one TEXTUAL✘ relationship in MaNGOS persisted for 334 days and included 676 changesets by one of its developers.

If developers know that they can merge others' changes safely, they may do so quickly and thus prevent a future conflict. The longer a TEXTUAL✓ relationship persists, the more opportunities it has to change into a conflict. Accordingly, we asked "How long do developers experience the TEXTUAL✓ relationship?" We measured the lifespan of a TEXTUAL✓ relationship for each conflict-free merge in the history (again comparing the changeset that introduced the relationship to the one that resolved it).

On average, the TEXTUAL✓ relationship persisted for 11 days and spanned 23 changesets — 11.5 per developer — (with median values of 1.9 days and 3 changesets) before incorporation (right side of Figure 6). A tool could have helped developers learn immediately about the TEXTUAL✓ relationship, encouraging earlier, smooth incorporation. In the worst case, in terms of time, one TEXTUAL✓ relationship in Voldemort persisted for 138 days; in terms of changesets, one TEXTUAL✓ relationship in Gallery3 persisted for 232 changesets without a merge, while each of the possible merges along the way would have been textually clean and fully automated.

4.4 Escalation of clean merges into conflicts

RQ3: Do clean merges devolve into conflicting changes?

Parallel work enables faster progress, but also the creation of conflicts. We, and others, argue that developers should perform safe merges as frequently as possible. To determine how often, in practice, parallel editing escalates into a conflict, we used a methodology similar to that of Section 4.3.

Every conflict relationship develops from a situation in which a second developer makes a change without having incorporated and understood a first developer's work. We found that of all conflict relationships (TEXTUAL✘, BUILD✘, and TEST✘), 93% developed from a TEST✓ relationship; the other 7% of conflict relationships developed from a BEHIND relationship. In other words, in almost every case, both developers had already committed (but not shared)

system	TEXTUAL✘ relationships							TEXTUAL✓ relationships						
	number	length (days)			length (changesets)			number	length (days)			length (changesets)		
		mean	stddev	median	mean	stddev	median		mean	stddev	median	mean	stddev	median
Voldemort	39	25.7	35.0	8.9	12.8	16.4	6	128	35.0	46.6	7.3	9.7	15.1	3
Gallery3	80	3.1	9.4	0.7	7.5	19.7	3	483	7.1	12.5	1.0	14.3	30.0	3
Insoshi	27	11.8	21.2	4.8	9.4	16.3	3	87	7.7	13.5	3.7	6.3	8.5	3
MaNGOS	58	8.2	44.0	1.8	17.6	83.0	3	118	2.4	2.1	1.7	5.8	7.1	3
Total	204	9.8	30.3	1.6	11.6	96.9	3	816	11.0	23.8	1.9	11.5	24.2	3

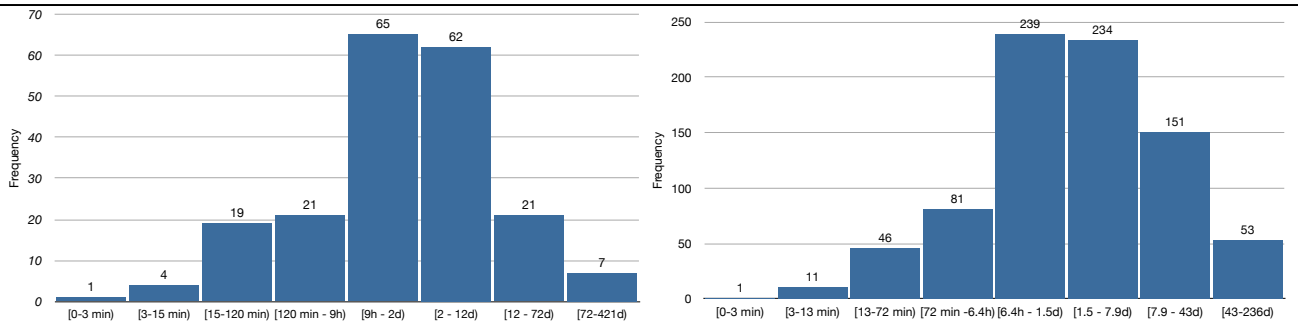

Figure 6: Persistence of the TEXTUAL✘ (left) and TEXTUAL✓ (right) relationships in historical data. The distributions are shown in eight-bucket geometric-progression histograms.

changes *before* the conflict developed. Every TEXTUAL✘ relationship could have been prevented by earlier incorporation of others' changes. Some BUILD✘ and TEST✘ relationships could also be prevented, and the others would at least be discovered locally and immediately, and would never be committed to the version control system.

We also found that 20% of TEST✓ relationships devolved into a conflict. The remaining 80% of TEST✓ relationships were merged successfully, preventing a conflict from developing. This suggests that what we call "safe merges" are actually at risk of devolving into conflicts that require human effort to resolve. Being aware of these merges early may prevent some such conflicts from arising.

While DVCSes record sufficient information to let us reconstruct how often a conflict arose from a BEHIND relationship, they do not record information that would allow us to determine how often a BEHIND relationship devolves into a conflict. We suspect that BEHIND relationships are also risky.

5. Information about conflicts

RQ4: What information could developers use to reduce the frequency and duration of conflicts?

What kind of unexploited information is available from a VCS that is not yet leveraged to smooth collaboration? What information could help the developer make better-informed decisions, such as whether to perform a particular VC operation, to wait for a coworker to perform one, or to communicate directly with a coworker?

We explored this space systematically by analyzing a hypothetical global view of all version control information in all repositories. We then systematically analyzed this perspective by (1) enumerating all possible version control situations that can arise during collaborative development, (2) determining, based on the global perspective, the best course of action for the team, (3) identifying what information that decision depended on, and (4) classifying the advice for the team.

We made the following assumptions in our exploration:

- Without loss of generality, we considered all situations with three developers, using the third developer to represent arbitrary other developers and repository hierarchies.
- We limit our lookahead to two rounds of version control operations: one developer performs one VC operation and then the other developer may or may not perform one.

This approach has allowed us to identify key common and important use cases.

Section 5.1 describes five pertinent local states of a developer. We augment our classification of the relationships between developers' repositories (already described in Section 3.2) with two other categories of information: the developer's possible actions (Section 5.2) and guidance about those actions (Section 5.3). Finally, we describe the information available about higher-order conflicts, specifically build and test conflicts (Section 5.4).

5.1 Local states

The five possible local states are:

uncommitted There are uncommitted changes in the working copy.

in conflict The local repository is in conflict with itself; that is, it has two heads that are not automatically mergeable. This happens, for example, when pulled changesets conflict with local changesets.

build failure The repository's version of the code fails to build.

test failure The repository's version of the code builds but fails its test suite.

OK The repository's version of the code builds and passes its test suite.

These states are not mutually incompatible; for example, a user may have uncommitted changes, be in conflict with itself, and have different build/compile status for each head and for the working copy. Furthermore, these states obscure some information, such as whether the working copy has been updated to all of the changesets in the local repository. The list also omits some states, such as when the local repository has two heads that can be merged automatically. Our approach and tools can handle such situations. For simplicity of exposition, however, this paper classifies each developer's state as the first one in the list that holds. This is all the information about state that is needed to provide the generally best advice to the team.

5.2 Actions

Given two repositories *A* and *B*, the relationship between *A* and *B* as well as the local states determine the possible actions that developers can perform.

SAME:	Nothing to do.
AHEAD:	May push; the new relationship will be SAME.
BEHIND:	May pull; the new relationship will be SAME.
TEXTUAL✘:	May pull; will result in the "in conflict" state. May push; *B* will be in the "in conflict" state.
BUILD✘:	May pull and merge; will result in the "build failure" state. May push; *B* will be in the "build failure" state.
TEST✘:	May pull and merge; will result in the "test failure" state. May push; *B* will be in the "test failure" state.
TEST✓:	May pull and merge; the new relationship will be AHEAD. May push; *B* will be able to merge the changes cleanly.

The consequences of applying available actions can be tricky to understand and remember. One example is when the available actions are the same but the consequences differ. For example, the developer can cleanly pull in both the BEHIND and TEST✓ relationships. However, in the BEHIND case, the developer ends up in the SAME relationship, while in the TEST✓ case, the developer ends up in the AHEAD relationship. Another example is when there are side-effects of performing an operation intended to change the relationship between *A* and *B*, such as incorporating *B*'s changes into *A* may put *A* and another repository *C* into a TEXTUAL✘ relationship. Using global version control information to help developers track such situations can be beneficial.

The local state also partially determines which version control operations can be executed in different situations.

If *A*'s state is "uncommitted", then the "update" operation cannot be applied. If *A*'s state is not "uncommitted", then the "commit" operation is inapplicable.

If *A*'s state is "in conflict", then all operations except "merge" are discouraged. (DVCSes permit most operations at any moment, but discourage some of them, most commonly by aborting the operation unless the user supplies an extra confirmation flag.) If *A*'s state is not "in conflict", then the "merge" operation is inapplicable. The "build failure" and "test failure" states do not limit the possible actions — VCSes are as yet unaware of such local states — although fixing these problems should likely be a priority. The "OK" state does not limit the possible actions.

5.3 Guidance

Information about how each action may affect the developer's state and relationships can help developers make better-informed decisions.

This section makes one common, generally realistic assumption: developers perform actions in a tree hierarchy, pushing only to and pulling only from a parent. This aligns with how developers predominantly interact with VCSes [31]. Further, we consider only information relevant to two developers who share a common parent repository (possibly that of one of the developers themselves), because in all other cases, the developers' relationship is dependent on actions by others.

We classify the guidance information into five types. One type of information concerns the relationship: *Committer*. The other four concern the possible action: *When*, *Consequences*, *Capable*, and *Ease*.

Committer: Who made the relevant changes?
Consider George and Ringo again, now working on a team with Paul. If George knows he is in the TEXTUAL✘ relationship with Ringo, George might decide to contact Ringo to discuss the situation. However, Ringo may not have made the conflicting changes; instead, Paul may have made and pushed the changes to the master, and Ringo then pulled them from the master. In this case, George should likely discuss the conflict with Paul rather than with Ringo. Knowing the committer facilitates communication between relevant parties, which in turn decreases the time required to fix conflicts [6].

When: Can an action that affects the relationship be performed now, or must it wait until later?
Ringo can be in the BEHIND relationship with George but may be unable to incorporate his changes because George may not have yet pushed them to the master. Thus, it may be helpful for Ringo to know that although he will need to pull at some point, he cannot get George's changes until George pushes. As another example, a developer may have to resolve an "in conflict" state before being allowed to push.

Consequences: Will an action — perhaps one on a different repository — affect a relationship?
The situation with Ringo BEHIND George illustrates this kind of guidance as well. Is Ringo BEHIND George because George has not yet pushed his changes to the master, or because George has pushed but Ringo has not yet pulled from the master? In the first case, even if Ringo pulls from the master, his relationship with George will not change. In the second case, if Ringo pulls, he will become SAME with George.

Capable: Who can perform an action that changes the relationship?
Consider a situation in which George is in the TEXTUAL✘ relationship with Ringo. If Ringo has already pushed his changes to the master, then George *must* be the one who resolves the conflict when he eventually pulls from the master. Conversely, if George has already pushed his changes, he *cannot* resolve the conflict any longer. And if neither has pushed, either of them *might* be the one to resolve the conflict.

Ease: Has anyone made changes that ease resolving an existing conflict?
Suppose George and Ringo created conflicting changes and Ringo has pushed his to the master. If George were to pull from the master, he would have to resolve the conflict. What if Ringo has made a set of follow-up changes that he has not yet pushed? If these changes resolve the conflict, then it is likely better for George to wait for Ringo to push his new changes. Ringo's pushing action would be the best way to resolve George's TEXTUAL✘ with the master.

By performing actions, developers can affect how long a conflict persists, or even prevent it from ever occurring. The guidance information can help developers decide which actions to perform. Knowing of a conflict relationship can encourage the developer to address it earlier, while the changes are fresh in the relevant developers' minds; this may reduce the conflict's duration as well as the effort necessary to resolve it. Knowing about BEHIND and TEST✓ relationships can reassure developers that it is safe to incorporate others' changes, which in turn keeps the development states closer together. In some cases, this may also allow the developer to prevent some potential conflicts altogether, which would also reduce conflict frequency. At a minimum, these relationships can prompt developers to communicate, which can reduce conflicts in the developers' mental models and work plans.

The *Committer* guidance informs the developers of who else is relevant to a conflict, reducing the time required to resolve it [6]. The *When* and *Capable* guidance can inform developers of the right time to perform an action, eliminating the overhead of manually figuring out if an action can be performed now and possibly having to

undo actions later. The *Consequences* guidance can allow the developers a peek into the future, also limiting undoing and redoing of work. Finally, the *Ease* guidance can inform a developer if someone else may have an easier time resolving a conflict, thus helping reduce the effort needed to resolve it.

5.4 Examples of higher-order conflicts

Early identification of higher-order conflicts between developers reduces — or at the least is highly unlikely to increase — the time to resolve a conflict. We describe a TEST✘ and a BUILD✘ example from Voldemort.

Malformed non-code resource: On October 10, 2009, a developer successfully merged two branches ("tips" in Git), `50b74` and `00c35`. Branch `00c35` was edited 17 times while the branch was alive and the last changeset on this branch occurred only eight minutes before the merge. Branch `50b74` had not been edited in the previous 48 days. Although the patch between these two branches was very large (63,413 lines), Git successfully merged these changesets. Test `voldemort.store.http.HttpStoreTest::testBadPort()` did not fail either branch before the merge, but did in the merged system. Thus, some unintended behavioral interaction between the two branches' changes broke this test. In fact, the merge invalidated one of the metadata files, `cluster.xml`. In this case, if a tool had let the developers know that it was safe to merge earlier, the problem could have been avoided completely by sequentializing the changes to `cluster.xml` and/or by enabling earlier testing of the merged version.

Missing type: On November 9, 2009, a developer successfully merged branches `c77a4` and `7f776`. Branch `7f776` was edited 11 times while the branch was alive; branch `c77a4` was edited three times. Both branches had been modified within four days of the merge. While the merge had no textual conflicts, the code failed to build: four compilation errors resulted from referencing a missing type `ProtoBuffAdminClientRequestFormat`. Later, the developer merged in another branch (`68e3b`), which resolved the compilation problem.

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

6. Delivering version control advice

Given that version control conflicts are frequent and serious (Section 4) and that a global view of the VCS could detect conflicts and reduce their frequency and severity (Section 5), how can a tool effectively deliver that information and advice to developers?

The design of our tool, Crystal, has several ways to convey the key information without overwhelming or distracting the developer, (1) The main window *summarizes* all projects and relationships, allowing a developer to instantly scan it to identify situations that may require attention. (2) The main window is *compact* but *not needed* if the developer prefer to receive limited but critical information. (The main nor any other window is ever opened asynchronously.) (3) Full, detailed information about each relationship, action, and guidance is *available but hidden* until a developer shows specific interest in it.

Crystal uses (1) icons exploiting color and shape in stable locations in the main window (rather than, say, a textual list that a developer would have to read and interpret), (2) a taskbar icon in the system tray to report the most severe state for all tracked repositories, and (3) mouse-over tooltips that provide, on demand, all the information discussed in Section 5.

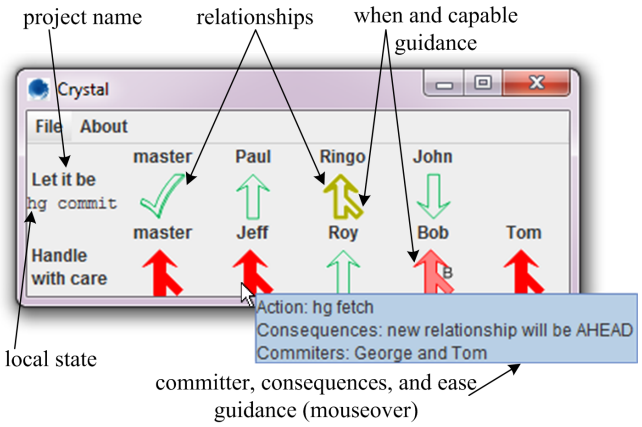

Figure 7: A screenshot of George's view of Crystal. George is following two projects under development: "Let it be" and "Handle with care". The former has four observed collaborators: George, Paul, Ringo, and John; the latter has five: George, Jeff, Roy, Bob, Tom. Crystal shows George's local state and his relationships with the master repository and the other collaborators, as well as guidance based on that information.

Crystal addresses scalability by allowing developers to select the repositories' relationships of interest. For example, a developer may be interested in the relationships with those in the same collaborative team and the main development repositories of the other teams. Crystal can provide information about relationships even with developers who are not using it, easing adoption by avoiding a requirement that the whole team uses the tool. Each developer can independently choose whether or not to run Crystal. Advantages accrue when more members of a team use Crystal, but this is not necessary.

Crystal allows developers to select a subset of the tests to execute, to integrate more smoothly into large development projects with extensive test suites. Naturally, for large projects with build scripts and test suites that take a long time to execute, Crystal will experience that latency. However, it would still identify relevant information sooner than other existing methods.

We have implemented our design in Crystal, a tool that performs speculative analysis of version control operations. Crystal currently works with the Mercurial DVCS; later versions of Crystal will support additional VCSes. Microsoft — in cooperation with us — is developing a version of Crystal. Crystal is an open-source, cross-platform, standalone tool and is available for download: `http://crystalvc.googlecode.com`. Our initial qualitative evaluation of Crystal is positive, but future work should evaluate it via both qualitative and quantitative user studies.

6.1 Crystal's UI

Figure 7 shows a screenshot of Crystal's main window. The window displays a row of icons (see Figure 8) for each of a developer's projects. In this example, there are two projects: "Let it be" and "Handle with care". The former has four collaborators: George (the developer running Crystal), Paul, Ringo, and John. The latter has five collaborators: George, Jeff, Roy, Bob, and Tom.

On the leftmost side of the row, underneath the project name, Crystal displays the local state. This tells George (in the native language of the underlying VCS) whether he must commit changes (`hg commit`, in Mercurial) or resolve a conflict. Then Crystal displays the relationship with the master and the collaborators' repositories.

If George has the ability to affect a relationship *now*, the icon is

Figure 8: Crystal associates an icon with each of the seven relationships. The color of each relationship icon represents the severity of the relationship (Section 3.2): relationships that require no merging are green, that can be merged automatically are yellow, and that require manual merging are red.

solid, which combines the *When* and *Capable* guidance. If George *cannot* affect the relationship, the icon is hollow. For example, consider John, who has made some changes in "Let it be" but has not yet pushed them to the master; George is BEHIND John, but the icon is hollow because George cannot affect this relationship until John pushes. Similarly, George's relationship with Ringo is a hollow TEST✓ because (1) George has the SAME relationship with the master, (2) Ringo had not pulled the latest changes from the master, and (3) Ringo has made some other changes, which he has not pushed but which can merge without human intervention. If the relationship is of the *might* variety — George might or might not have to perform an operation to affect the relationship — the icon is solid but slightly unsaturated (see the relationship with Bob in the "Handle with care" project).

These features allow George to quickly scan the Crystal window and identify the most urgent issues, the solid red icons, followed by other, less severe icons. George can also easily identify quickly whether there is something he can do now to improve his relationships (in the example, George can perform actions to improve his relationships in the "Handle with care" project, but not in "Let it be"), and whether there are unexpected conflicts George may wish to communicate with others about.

The most urgent relationship is displayed by Crystal as its system tray icon, which allows a developer to know at all times whether there is any action that requires attention without even having the Crystal window open.

Crystal also provides other guidance that is hidden unless a developer wants to see it. Holding the mouse pointer over an icon displays the action George can perform and the *Committer*, *Consequences*, and *Ease* guidance, when applicable. For example, when George holds the mouse over Jeff's TEXTUAL✘ icon, it tells him that he can perform a pull and a resolve (`hg fetch`, in Mercurial), that performing this action will resolve George's TEXTUAL✘ with Jeff, and that Tom and George committed the conflicting changesets.

Even though George asked for information about the relationship with Jeff, Crystal was able to correctly point George to Tom as the developer who was responsible for the conflicting changesets (which Jeff had pulled into his repository). In other situations, it is possible that George performing a pull and resolve operations with his parent would not resolve George's TEXTUAL✘ with Jeff (e.g., if Jeff and Tom had both created conflicting changesets but only Tom had pushed his changesets to the master). This is why the consequences guidance is important. As a final note, because no one else has merged these changesets, George must resolve this conflict and there is no *Ease* guidance for Crystal to display.

6.2 Initial experience

Crystal consists of 5,200 NCSL of Java and has been tested on Windows, Mac OS X, and several Linux distributions. The developer using Crystal must have read access to the collaborators' repositories; the Crystal manual (available at `http://crystalvc.googlecode.com`) describes several simple ways to accomplish this.

We deployed the beta-test version of the tool to a small number of developers and have been using it ourselves, and refining it, since early July 2010. One co-author uses Crystal to monitor 49 clones of 10 projects belonging to eight actively working collaborators.

Designing and deploying Crystal, along with frequent feedback from the handful of users, has helped us to better understand the issues and to improve the tool's design. Crystal user feedback enhanced our understanding of the need for guidance as well as which information is most pertinent to make available to the developer. For example, showing hollow and solid icons arose from a user's need to differentiate between relationships he or she could and could not affect. The feedback drove us to systematically explore the complete space, as described in Section 5.

Here is one example piece of feedback from an external user, via private communication:

> "Keeping a group of developers informed about the state of a code repository is a problem I have tried solving myself. My solution was an IRC bot that announced commits to an IRC channel where all of the developers on the project idled. This approach has many obvious problems. [...] The Crystal tool does not suffer from these problems. Crystal handles several projects and users effortlessly and presents the necessary information in a simple and understandable way, but it is only a start at filling this important void the in the world of version control."

Prior to developing Crystal, we surveyed 50 DVCS users about their collaborative development habits. Their use of highly heterogeneous operating systems, IDEs, VCSes, and languages informed Crystal's design. Even among this small group, there were vast differences in committing, pushing, and pulling styles, which further encouraged our research. We anticipate that future user studies will identify additional strengths and weaknesses that will allow us to further improve Crystal.

7. Related Work

This section places our research in the context of related work in evaluating the costs of conflicts, collaborative awareness, mining software repositories, and continuous development.

7.1 The cost of conflicts

Efficient coordination is important for effective software development. The number of defects rises as the amount of parallel work increases [23], but developers can more effectively manage risks to the consistency of their systems if they are aware of the consequences of their commits on other developers [11]. Developers eschew parallel work to avoid having to resolve conflicts when committing changes [13], or rush their work into the trunk to avoid being the developer who would have to resolve conflicts [8].

Several observational and laboratory experiments empirically assess the benefits of collaborative awareness for configuration management [3, 10, 30]. Augmenting these results, we performed a retrospective analysis on real projects to estimate the potential benefit. Our analysis is consistent with their studies in confirming the potential for better coordination of individual and team repositories.

Sarma provides a comprehensive classification of collaborative tools for software development [27]; in this classification, Crystal could be considered a seamless tool as it provides continuous awareness about development state and guidance about the consequences of potential future actions.

7.2 Collaborative awareness

The research most similar in intent to ours studies collaborative awareness — increasing awareness of the activities among team

members. Such awareness can be a distraction unless a conflict is imminent, so awareness tools have adopted increasingly sophisticated methods for avoiding false positive warnings, as we now describe.

Palantír [29, 28, 1] shows which developers are changing which artifacts by how much. Palantír has similar motivations to ours: "providing 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" [29, p. 444]. FASTDash [3] is similar: it is an interactive visualization — a spatial representation of which files each developer is editing — that augments existing software development tools with a specific focus on helping developers understand what other team members are doing.

Syde [15] reduces false positives via a fine-grained analysis of the abstract syntax trees (ASTs) modifications. Two potentially conflicting changes to the same file are flagged for a developer only when they also affect changes to the same parts of the underlying ASTs. For example, if two users have inserted, deleted, or changed the same method, the changes will be flagged "yellow"; if one of user had committed, the changes would instead be flagged "red", indicating that there may be a merge conflict. Syde examines files every time they are saved.

The most detailed analysis is done by tools like CollabVS and Safe-commit. CollabVS "detects a potential conflict when a user starts editing a program element that has a dependency on another program element that has been edited but not checked-in by another developer" [10]. Safe-commit [33] does the deepest program dependence analysis, identifying changes that are guaranteed not to cause tests to fail. This allows earlier publishing of some of a developer's changes, on the theory that increasing the publishing frequency can decrease the amount of duplicate development and the likelihood of merge conflicts.

Our approach suffers fewer false positives and fewer false negatives than previous awareness approaches [5] for several reasons. First, our approach computes actual pending conflicts rather than estimating potential ones. By speculatively doing exactly what a developer will actually do in the future — run a version control operation, then run the build script and, finally run the test script — our approach only reports problems that would actually happen while executing those steps. (A secondary benefit of using the underlying VCS directly is that users of Crystal can benefit immediately from any improvements to the VCS merging algorithm.) Second, Crystal does not report conflicts until they have been committed to some repository. This reduces false positives resulting from exploratory edits, such as for debugging: developers typically commit code that is consistent and is a candidate for sharing. This could delay delays Crystal's reports until a commit occurs, but commits tend to be frequent in a DVCS. Third, unlike most of the previous work, our approach aids developers in performing safe merges earlier, in addition to early isolation of conflicts. Fourth, also unlike most of the previous work, we consider and support multiple levels of conflicts — textual, build, and test.

7.3 Mining software repositories

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. Later efforts mine version histories to determine functions that must likely be modified as a group [35], to identify common error patterns [19], to predict component failures [21], etc.

Our effort contrasts with these efforts in at least two dimensions. First, we are assessing a different property: opportunities to incorporate changes with others on a team. Second, the purpose of our mining was to determine whether building a tool like Crystal would be worthwhile. Other mining efforts generally aim to improve a team's software development process, such as by informing managers of a pattern so that they will allocate more quality assurance resources to more error-prone components.

7.4 Continuous development

Our approach can be characterized as continuous merging. Thus, it is related to a number of other approaches to continuous computation in the context of software development.

A programming environment, modeled on spreadsheets, can continuously execute the program as it is being developed [16, 18]. Modern programming environments focus instead on providing 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 (and usually unobtrusive) feedback about compilation errors, allowing for quick correction while that code is fresh in the developer's mind. Second, the developer is freed from deciding when to compile, meaning that the developer is not distracted by the compilation task and that when it is time to run or test the code, no intervening compilation step is necessary.

Continuous testing [24, 25, 26, 12] applies the same idea to testing: it 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. Similar ideas are gaining traction in the development community.

Recent work has investigated supporting real-time integration to decrease developer's hesitation in committing changes using centralized version control [14]. Like FASTDash, this approach aims to help developers avoid conflicts but, in contrast to FASTDash and similarly to Crystal, it computes rather than predicts the presence of merge conflicts.

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

8. Threats to validity

Our research, naturally, leaves open a set of potential concerns, which we present using the standard notions of threats to validity.

Construct.

The version control histories tell us when a TEXTUAL✘ or TEXTUAL✓ relationship first arose and when the developers resolved it. However, the histories do not tell us (1) when or how the developers found out about the relationship, (2) when the developers began trying to resolve the relationship, and (3) had the developers known about the relationship earlier, would they have done anything differently?

In addition, DVCS histories only contain information about incorporate operations from the TEXTUAL✓ and TEXTUAL✘ states; nothing is recorded when a developer pulls from the BEHIND state or pushes from the AHEAD state.

Internal.

Our experiments (Section 4) are in the context of DVCSes, which differ from CVCSes [7, 20, 22]. The effect of the VCS on developer behavior is not established: indeed, various researchers hypothesize the full spectrum — that a DVCS causes a developer to publish less often, more often, or the same amount as a CVCS. If DVCSes encourage more frequent branching and merging [32], that would provide additional opportunities for Crystal.

While our full retrospective analysis cannot be done on the histories of repositories built using existing CVCSes, we believe the data we find in DVCS projects is an approximation of what happens in development with CVCSes; the largely similar results to Zimmermann [34] justifies this belief.

External.

Another threat is that our retrospective study focused on nine open-source systems. The systems we selected may not be characteristic of other systems. Anecdotally, developers are all-but-universally worried about the problems that can arise from conflicts. The professional web (blogs, Q&A sites, etc.) is filled with examples of developers expressing this concern and suggesting ways to reduce it.

Usability and developer style.

While Crystal can answer important questions about the developers' relationships in a collaborative environment and aid those developers in making better-informed decisions, Crystal might also harm productivity by distracting developers or leading them to premature integration. To mitigate the issues of distraction, we have worked to reduce Crystal's intrusiveness. In particular, humans tend to be reasonably good at selecting which information to ignore, and we have designed Crystal to be consistent with that ability. Some developers may prefer to use the full Crystal view, while others may prefer the system-tray view most of the time. And a developer who is "heads-down" can simply quit Crystal for a while, just as many developers choose to, at times, ignore their email.

One challenge to Crystal's adoption may be that developers may fail to see its utility. One developer who attempted to use Crystal reported that he simply was not interested in seeing conflicts with unpublished changes and that he rarely experienced conflicts with others in his development. While he saw no harm to running Crystal, he anticipated it would provide him no benefit either. Crystal, indeed, may well not be appropriate for all classes of developers. Nonetheless, the data in Section 4 show that conflicts are common, not rare, suggesting strongly that most developers may well benefit from Crystal, regardless of their intuitions. We plan to test this hypothesis as part of a future user study.

Furthermore, conflicts are not the only reason to use Crystal. The developer who declined to use Crystal ended up doing redundant work. He noticed a problem and fixed it — but another developer had already made the same fix, and pushed it, six days earlier. The non-Crystal-user had forgotten to pull changes before beginning to work on the problem. Crystal would have reminded the user that he could pull changes, and had he followed Crystal's advice, he would have avoided the wasted effort of the duplicated bug fix.

9. Conclusions

Speculative analysis over version control operations provides precise information about pending conflicts between collaborating team members. These pending conflicts — including textual, build, and test — are guaranteed to occur (unless a developer modifies a committed change). Learning about them earlier allows developers to make better-informed decisions about how to proceed, whether it be to perform a safe merge, to publish a safe change, to quickly address a new conflict, to interact with another developer etc.

Our retrospective, quantitative study of over 550,000 development versions of nine open-source systems, spanning 3.4 million distinct (and a total of over 500 billion, over all versions) NCSL, confirms that (1) conflicts are the norm rather than the exception, (2) that 16% of all merges required human effort to resolve textual conflicts, (3) that 33% of merges that were reported to contain no textual conflicts by the VCS in fact contained higher-order conflicts, and (4) that conflicts persist, on average, for 10 days (with a median conflict persisting 1.6 days). Although there is a significant amount of qualitative and anecdotal evidence consistent with our findings, the only previous quantitative research we could find was Zimmermann's [34]. We expand on his work in several dimensions, including (1) comparing actual merges from project histories to merges that could have taken place successfully earlier than they did, and (2) considering not only textual conflicts but also higher-order conflicts, such as build and test conflicts.

Our speculative analysis tool, Crystal, provides concrete information and advice about pending conflicts while remaining largely unobtrusive. Our evaluation of Crystal is preliminary and qualitative; future work should evaluate it via both qualitative and quantitative user studies.

Collaborative development is essential but troublesome. Making pertinent and precise information available to developers, allowing them to identify and fix conflicts before they fester, is one useful and practical step in reducing some of the costs and difficulties of collaborative software development.

Acknowledgments

The Crystal beta users provided valuable feedback. This material is based upon work supported by the National Science Foundation under Grants CNS-0937060 to the Computing Research Association for the CIFellows Project and CCF-0963757, by a National Science and Engineering Research Council Postdoctoral Fellowship, and by Microsoft Research through a Software Engineering Innovation Foundation grant.

References

[1] Ban Al-Ani, Erik Trainer, Roger Ripley, Anita Sarma, André van der Hoek, and David Redmiles. Continuous coordination within the context of cooperative and human aspects of software engineering. In *CHASE*, pages 1–4, Leipzig, Germany, May 2008.

[2] Thomas Ball, Jung-Min Kim, Adam A. Porter, and Harvey P. Siy. If your version control system could talk... In *PMESSE*, Boston, MA, USA, May 1997.

[3] Jacob T. Biehl, Mary Czerwinski, Greg Smith, and George G. Robertson. FASTDash: A visual dashboard for fostering awareness in software teams. In *CHI*, pages 1313–1322, San Jose, CA, USA, Apr. 2007.

[4] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Speculative analysis: Exploring future states of software. In *FoSER*, pages 59–63, Santa Fe, NM, USA, Nov. 2010.

[5] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Crystal: Precise and unobtrusive conflict warnings. In *ESEC FSE Tool Demo*, Szeged, Hungary, Sep. 2011.

[6] Marcelo Cataldo, Patrick A. Wagstrom, James D. Herbsleb, and Kathleen M. Carley. Identification of coordination requirements: Implications for the design of collaboration and awareness tools. In *CSCW*, pages 353–362, Banff, AB, Canada, Nov. 2006.

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

[8] Cleidson R. B. de Souza, David Redmiles, and Paul Dourish. "Breaking the code", Moving between private and public work in collaborative software development. In *GROUP*, pages 105–114, Sanibel Island, FL, USA, Nov. 2003.

[9] Prasun Dewan. Dimensions of tools for detecting software conflicts. In *RSSE*, pages 21–25, Atlanta, GA, USA, Nov. 2008.

[10] Prasun Dewan and Rajesh Hegde. Semi-synchronous conflict detection and resolution in asynchronous software development. In *ECSCW*, pages 159–178, Limerick, Ireland, Sep. 2007.

[11] Jacky Estublier and Sergio Garcia. Process model and awareness in SCM. In *SCM*, pages 59–74, Oxford, England, UK, Sep. 2005.

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

[13] Rebecca E. Grinter. Using a configuration management tool to coordinate software development. In *CoOCS*, pages 168–177, Milpitas, CA, USA, Aug. 1995.

[14] Mário Luís Guimarães and António Rito-Silva. Towards real-time integration. In *CHASE*, pages 56–63, Cape Town, South Africa, May 2010.

[15] Lile Hattori and Michele Lanza. Syde: A tool for collaborative software development. In *ICSE Tool Demo*, pages 235–238, Cape Town, South Africa, May 2010.

[16] Peter Henderson and Mark Weiser. Continuous execution: The VisiProg environment. In *ICSE*, pages 68–74, London, England, UK, Aug. 1985.

[17] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM TOPLAS*, 11:345–387, July 1989.

[18] R. R. Karinthi and M. Weiser. Incremental re-execution of programs. In *SIIT*, pages 38–44, St. Paul, MN, USA, June 1987.

[19] Benjamin Livshits and Thomas Zimmermann. DynaMine: Finding common error patterns by mining software revision histories. In *ESEC FSE*, pages 296–305, Lisbon, Portugal, Sep. 2005.

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

[21] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *ICSE*, pages 452–461, Shanghai, China, 2006.

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

[23] Dewayne E. Perry, Harvey P. Siy, and Lawrence G. Votta. Parallel changes in large-scale software development: an observational case study. *ACM TOSEM*, 10:308–337, July 2001.

[24] David Saff and Michael D. Ernst. Reducing wasted development time via continuous testing. In *ISSRE*, pages 281–292, Denver, CO, USA, Nov. 2003.

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

[26] David Saff and Michael D. Ernst. An experimental evaluation of continuous testing during development. In *ISSTA*, pages 76–85, Boston, MA, USA, July 2004.

[27] Anita Sarma. A survey of collaborative tools in software development. Technical Report UCI-ISR-05-3, University of California, Irvine, Institute for Software Research, 2005.

[28] Anita Sarma, Gerald Bortis, and André van der Hoek. Towards supporting awareness of indirect conflicts across software configuration management workspaces. In *ASE*, pages 94–103, Atlanta, GA, USA, Nov. 2007.

[29] Anita Sarma, Zahra Noroozi, and André van der Hoek. Palantír: raising awareness among configuration management workspaces. In *ICSE*, pages 444–454, Portland, OR, May 2003.

[30] Anita Sarma, David Redmiles, and André van der Hoek. Empirical evidence of the benefits of workspace awareness in software configuration management. In *FSE*, pages 113–123, Atlanta, GA, USA, Nov. 2008.

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

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

[33] Jan Wloka, Barbara Ryder, Frank Tip, and Xiaoxia Ren. Safe-commit analysis to facilitate team software development. In *ICSE*, pages 507–517, Vancouver, BC, Canada, May 2009.

[34] Thomas Zimmermann. Mining workspace updates in CVS. In *MSR*, Minneapolis, MN, USA, May 2007.

[35] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *ICSE*, pages 563–572, Edinburgh, Scotland, UK, 2004.