

Early Detection of Collaboration Conflicts and Risks

Yuriy Brun, *Member, IEEE*, Reid Holmes, Michael D. Ernst, and David Notkin, *Fellow, IEEE*

Abstract—Conflicts among developers' inconsistent copies of a shared project arise in collaborative development and can slow progress and decrease quality. Identifying and resolving such conflicts early can help. Identifying situations which may lead to conflicts can prevent some conflicts altogether. By studying nine open-source systems totaling 3.4 million lines of code, we establish that conflicts are frequent, persistent, and appear not only as overlapping textual edits but also as subsequent build and test failures. Motivated by this finding, we develop a *speculative analysis* technique that uses previously-unexploited information from version control operations to precisely diagnose important classes of conflicts. Then, we design and implement Crystal, a publicly-available tool that helps developers identify, manage, and prevent conflicts. Crystal uses speculative analysis to make concrete advice unobtrusively available to developers.

Index Terms—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* [13], [17], [19], [25], [32], [47] 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 [13], [19]. Ironically, this fear of *potential* conflicts can cause developer copies to diverge even further, 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 prevents the second developer from publishing until the conflict is resolved automatically (by the VCS) or manually (by a developer). Higher-order conflicts arise when there are no textual conflicts among

developers' changes, but those changes are semantically incompatible. Higher-order conflicts cause compilation errors, test failures, or other problems, and are problematic to detect and resolve in practice [25].

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 [14].

Our approach, speculative analysis, unobtrusively provides information about the presence or absence of conflicts in a continuous and accurate way. We intend 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. This paper makes the following contributions:

- We analyze nine open-source systems. Conflicts between developers' copies of a project (1) are the norm, rather than the exception, (2) persist, on average, 3 days, and (3) are higher-order 33% of the time. (We make public and open-source our analysis tools and data.)
- We introduce a novel technique called speculative analysis that anticipates actions a developer may wish to perform and executes them in the background. When applied to collaborative development and version control systems, speculative analysis can use previously-unexploited information to precisely diagnose important classes of conflicts and offer concrete advice about addressing them. Reporting the consequences of these likely version control operations can improve the way in which collaborating developers identify and manage conflicts.
- We design and implement an open-source, publicly-available tool called Crystal — <http://crystalvc.googlecode.com> — that implements the analyses and unobtrusively presents advice to developers, to aid them in identifying, managing, and preventing conflicts. (See Figure 10 in Section 6 for a detailed Crystal screenshot).

• Y. Brun is with the School of Computer Science, University of Massachusetts, 140 Governors Dr., Amherst, MA 01003-9264.
E-mail: brun@cs.umass.edu.

• R. Holmes is with the David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada.
E-mail: rtholmes@cs.uwaterloo.ca

• M.D. Ernst and D. Notkin are with Computer Science & Engineering, University of Washington, PO Box 352350, Seattle, WA, 98195, USA.
E-mail: mernst@cs.washington.edu

Manuscript received 18 Nov. 2012; revised 30 Apr. 2013; accepted 10 May 2013; published online 24 May 2013.

Recommended for acceptance by A. Zeller.

For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2012-11-0334. Digital Object Identifier no. 10.1109/TSE.2013.28.

We have previously proposed how speculative analysis may help developers [7] and shown how it can improve recommendation systems [29]. Here, we extend an earlier version of this paper [9] in three ways: (1) Expanding our experimental, retrospective analysis of conflict lifespan (Section 4). (2) Exhaustively enumerating the space of collaborative relationships, actions developers may perform, and guidance regarding those actions (Sections 5.2 and 10). (3) Augmenting the comparison of related research to our work (Section 7).

Section 2 provides a brief scenario of two collaborating developers, sketching how their development activities would differ with 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, in his individual copy. When finished, George and Ringo each independently run the test suite on their individual copies (the tests pass for both of them), then publish their changes to the master repository. When the regression tests run after both have published, George and Ringo are notified that a test fails. 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, suppose George and Ringo were using a speculative analysis tool such as our tool Crystal, which proactively informs developers of version control conflicts. Crystal informs them before they publish their changes that integrating those changes would cause the test suite to fail (Figure 1). The tool encourages George and Ringo to address the impending conflict before they forget the relevant changes and assumptions.

Speculative analysis [7] neither guesses at possible conflicts nor approximate them. Instead, it speculatively performs the work, including 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

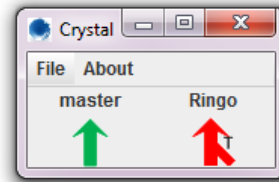


Fig. 1. A screenshot of the Crystal tool as run by a developer named George. The green arrow informs George that his changes can be published (uploaded) without conflict to the master repository. The red merge symbol indicates that Ringo's changes, if combined with George's, would cause a test ("T")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

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.

The frequency with which speculative analysis executes is adjustable. For example, merging code between two developers can happen whenever either of those developers make a change, at a regular time period, or a combination of the two. (By default, Crystal runs its analysis after checking, every 10 minutes, if new changes have been made and committed to a developer's local repository.) Running the analysis more frequently requires more computation, but produces conflict information sooner.

To handle exploratory development, our approach assumes that when a developer commits code to the VCS, the developer has decided to share that code with other developers at a future time. This can prevent some false warnings present in awareness systems when developers make exploratory changes that they never intend to share, such as adding print statements for debugging purposes. 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 (Section 5.4 will discuss how our approach applies to CVCS). 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 [44] 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

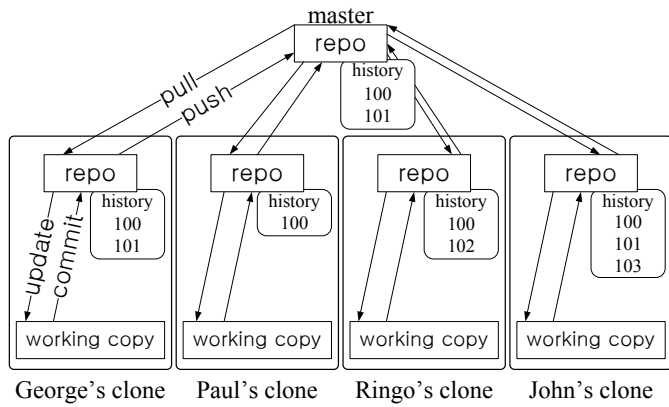


Fig. 2. A DVCS with four clones of a master repository. The box labeled “history” lists those changesets currently in the repository. The commit command creates a new changeset in its repository’s history, and the update command incorporates changesets into the working copy. A developer can incorporate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

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. A *merge* conflict can arise due to a pull operation, and the conflict must be *resolved* before proceeding. It is uncommon for a developer to pull changesets without immediately resolving (if necessary) and updating their working copy. The terms used above are common, or have direct equivalents, across DVCS systems.

The discussion in this paper makes two simplifying assumptions for clarity: (1) it assumes that developers push to and pull from only the master repository; (2) it assumes that developers only make a commit when all their tests pass. However, our approach and the Crystal tool 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 proper 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✗.

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 are relationships BUILD✓ = TEST✓ ∪ TEST✗ and TEXTUAL✓ = BUILD✓ ∪ BUILD✗. The table header of Figure 4 illustrates the interrelation among the relationships. When build scripts and test suites are not available, we 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 three research questions: First, Sections 4.1 and 4.2 address “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 focuses on BUILD✗, TEST✗, and TEST✓ relationships.) Second, Section 4.3 answers “How long do developers experience the conflict relationship TEXTUAL✗?” Third, Section 4.4 answers “How risky 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

system	KNCSL	devs	changesets	days	first version	last version	description
Gallery3	57	24	4,838	437	fff10f8	36702b1	Web-based photo album
Git	267	27	20,785	1,741	e83c516	7e94805	Version control system
Insoshi	173	15	1,316	629	189a6c6	7013235	Social networking platform
jQuery	26	23	2,183	1,393	8a4a1ed	8404ad6	JavaScript library
MaNGOS	643	27	3,511	626	b5bf407	acaaac7	Online game server
Perl5	660	51	34,653	8,061	8d063cd	6e3b7bf	Programming language
Rails	141	50	12,342	1,875	db045db	dc3cc6c	Web application framework
Samba	1,363	59	58,802	5,001	97e3422	0c39fbc	File and print services
Voldemort	103	22	1,219	375	fb0f95	0852fcb	Structured storage system
total	3,433	298	138,549	20,138			

Fig. 3. Nine subject programs analyzed to address RQ1, RQ2, and RQ3 in collaborative development environments. KNCSL stands for thousands of non-comment source lines.

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."

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 [47]. He reported that of all merges, 23% to 47% had textual conflicts (TEXTUAL✗) while the remainder could be merged automatically (TEXTUAL✓). Answering our first two research questions requires analyses that significantly augment these data and anecdotes.

In answering those research questions, and to augment Zimmermann's data and determine the validity of the anecdotal evidence, we performed an analysis on a set of open-source software projects. As subjects, (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 CVCS repository (which would not contain sufficient information to answer our research questions). For each of the projects, we used the version control history up to February 13, 2010.

The tools we created to perform the analyses described in this section are open-source. These tools and all our data are publicly available at <https://github.com/rtholmes/crystal-retrospective-analysis/>.

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 16%, 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 Crystal, for 19% of the commits, Crystal would have informed those developers about TEXTUAL✗ relationships. 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, 16% 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 shows 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. That is, 33% of the 399 clean merges led to build or test conflicts.

Few current awareness tools detect higher-order conflicts (see Section 7). Rather, they generally notify developers of

system	merges	TEXTUAL✗		TEXTUAL✓					
				BUILD✗		BUILD✓			
						TEST✗	TEST✓		
Git	1,362	227	17%	2	0.1%	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	506	47	9%			459	91%		
Insoshi	93	23	25%			70	75%		
jQuery	18	1	6%			17	94%		
MaNGOS	194	81	42%			113	58%		
Rails	362	51	14%			311	86%		
Samba	748	100	13%			648	87%		
total	3,635	572	16%			3,065	84%		

Fig. 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✓.

all changes to the repository (e.g., FASTDash [4]) or of concurrent changes to ASTs (e.g., Syde [23]). In contrast, we adopt 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 textual 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, we traced backwards in time through the history from the two changesets that were eventually merged to find the earliest point in time when the two branches came into conflict with each other. To do this, we created a time-ordered list of the changesets from each of the two branches and compared all distinct pairs of changesets that co-existed at each point in time to see if they were in conflict, stopping when we found a non-conflicting pair. This approach flattened all other sub-branches and merges that existed on the branches that contributed to the merge under analysis. (We omitted all conflicts between changesets that were never actually merged in the history, such as those on dead-end branches.)

On average, the TEXTUAL✗ relationship persisted for 3.2 days and involved 18.3 changesets (with median values of 0.7 days and 6 changesets) before being resolved (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 before it was resolved.

If developers know that they can merge others’ changes safely, they may choose to do so quickly and thus prevent a future

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%

Fig. 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.

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 starting with the changeset that introduced the relationship and ending with the one that resolved it).

On average, the TEXTUAL✓ relationship persisted for 2.4 days and involved 12.7 changesets (with median values of 0.8 days and 7 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. Neither of these two long-lived TEXTUAL✓ relationships evolved into a conflict.

4.4 Escalation of clean merges into conflicts

RQ3: Do clean merges devolve into conflicts?

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. 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. How often does parallel editing escalate into a conflict, in practice?

Using a methodology similar to that of Section 4.3, we found that 93% of the TEXTUAL✗ relationships developed from a TEXTUAL✓ relationship; the other 7% of developed from a BEHIND relationship. In other words, in almost every case, both developers had already committed (but not shared) changes *before* the conflict developed. Every TEXTUAL✗ relationship between repository commits can be prevented by incorporating others’ changes earlier. (In some cases, a developer may have to change his or her plans based on edits by others, and may need

system	#	TEXTUAL✗ relationships						#	TEXTUAL✓ relationships					
		length (days)			length (changesets)				length (days)			length (changesets)		
		mean	stddev	median	mean	stddev	median		mean	stddev	median	mean	stddev	median
Voldemort	28	7.5	9.9	2.1	23.2	33.1	6	139	6.0	13.6	2.7	16.0	35.4	7
Gallery3	47	1.8	6.9	0.1	20.8	70.0	4	459	1.1	6.9	0.6	11.2	36.3	6
Insoshi	23	7.3	14.7	1.9	24.3	44.4	6	70	4.9	13.4	1.1	10.8	13.1	6.5
jQuery	1	5.4	0	5.4	13.0	0	13	17	1.1	2.1	0.4	4.9	3.7	4
MaNGOS	81	1.4	1.5	0.8	13.5	14.0	9	113	2.4	2.8	1.5	16.7	19.6	11
total	180	3.2	8.0	0.7	18.3	42.6	6	797	2.4	7.7	0.8	12.7	32.5	7

Fig. 6. Persistence of the TEXTUAL✗ (left) and TEXTUAL✓ (right) relationships in historical data.

to make edits to resolve conflicts, but at least the conflicts would never be committed to the VCS.)

We also found that 20% of TEXTUAL✓ relationships devolved into a conflict. The remaining 80% of TEXTUAL✓ 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?

Multiple factors can affect the frequency and duration of conflicts. For example, the order in which developers incorporate changes may affect whether they encounter a conflict at all. Further, which developers communicate, and when they do so, may affect how quickly the developers can resolve a conflict.

Consider a scenario with three developers, George, Jeff, and Tom, collaborating on a project using a master repository. (These developers and their work are part of a larger scenario we will describe in Section 6.1 and Figure 10). Figure 7 shows the developers’ repository histories. All three create changes, but only George’s and Tom’s (101 and 103) conflict. Tom shares his changes with the master and Jeff incorporates that change from the master. At this point in time, George’s and Jeff’s repositories conflict. If George and Jeff discover that their repositories conflict, they may attempt to resolve the conflict together. However, that is not the ideal action for them to take because Jeff did not write any of the responsible code. It would be best for George to communicate with Tom, who is likely to be more familiar with the relevant changes than Jeff is.

This section explores and enumerates the space of collaborative relationships, actions developers may perform, and guidance regarding those actions. First, Section 5.1 enumerates the space. Then, Section 5.2 describes the exhaustive approach by which we enumerated the space. Section 5.3 augments the space with information specific to higher-order, build and test conflicts.

Finally, Section 5.4 describes differences between CVCS and DVCS that affect the space.

5.1 Available information

This section enumerates the space of collaborative relationships, actions developers may perform, and guidance regarding those actions. Section 5.1.1 describes five local states of a developer’s repository and working copy. Sections 5.1.2 and 5.1.3 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 and guidance about those actions.

5.1.1 Local states

A developer’s local state is information that can be obtained without querying any other repository. 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 working copy may have uncommitted changes at the same time that the repository is in conflict with itself and has a different build status for each head. 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.

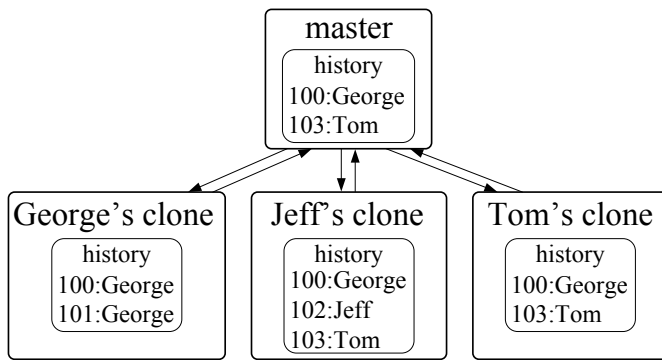


Fig. 7. A DVCS snapshot of three developers working in parallel. Each changeset in the history is annotated by who created it. Changesets 101 and 103 conflict textually, so George is in **TEXTUAL \times** with Jeff and with Tom.

5.1.2 Actions

Given two repositories A and B , the possible actions that developer A can perform depends on both the local state and the relationship between A and B .

Local states: The local state 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.

Repository relationships: In this section, we assume that the local state is neither "uncommitted" nor "in conflict", per DVCS best practices as discussed immediately above.

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

relationships. However, in the **BEHIND** case, the developer ends up in the **SAME** relationship, while in the **TEST \checkmark** 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 . For example, incorporating B 's changes into A may put A and another repository C into a **TEXTUAL \times** relationship. Using global version control information to help developers track such situations can be beneficial.

5.1.3 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: repositories are organized in a tree hierarchy, so developers only push to and pull from a parent. This aligns with how developers predominantly interact with VCSes, even DVCSes [44]. 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' future 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*.

Section 5.2 will enumerate the space of possible collaborative situations. For these situations, these five types of guidance information are sufficient for the developers to make the optimal choice in avoiding and resolving conflicts, given the information contained in the VCS we considered. Of course, this is not *all* the information available in a VCS that might be relevant to collaboration and to identifying and resolving conflicts. For example, each commit contains a descriptive, developer-written, natural-language message describing the commit's changes. This message may describe side-effects and could theoretically lead to a better understanding of a conflict. Further, information outside of the VCS can also affect conflict resolution, including the developers' organization and development policies. This information can further inform and improve conflict detection and resolution, but we do not study it here.

Committer: Who made the relevant changes?

Consider George, Jeff, and Tom again, from Figure 7. If George knows he is in the **TEXTUAL \times** relationship with Jeff, George might decide to contact Jeff to discuss the situation. However, Jeff did not make the conflicting changeset 103, Tom did and Jeff only incorporated Tom's change. In this case, George should likely discuss the conflict with Tom rather than with Jeff. Knowing the committer facilitates communication between relevant parties, which in turn decreases the time required to fix conflicts [10].

When: Can an action that affects the relationship be performed now, or must it wait until later?

Tom can be in the **BEHIND** relationship with Jeff but may be unable to incorporate his change (changeset 102) because Jeff has not yet shared it with the master. Thus, it may be helpful for

Tom to know that although he will need to incorporate at some point, he cannot get Jeff's change until Jeff shares it. 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 Tom BEHIND Jeff illustrates this kind of guidance as well. Is Tom BEHIND Jeff because Jeff has not yet shared his change (changeset 102) with the master, or because Jeff has shared with the master but Tom has not yet incorporated from the master? In the first case, even if Tom incorporates from the master, his relationship with Jeff will not change. In the second case, if Tom incorporates, he will become SAME with Jeff.

Capable: Who can perform an action that changes the relationship?

Consider a situation in which George is in the TEXTUAL \times relationship with Tom. Tom has already shared his change (changeset 103) with the master, so George *must* be the one who resolves the conflict when he eventually incorporates from the master. Conversely, if George had shared his change (changeset 101) first, he *could not* resolve the conflict. And if neither had shared, either of them *might* be the one to resolve the conflict.

Ease: Has anyone made changes that ease resolving an existing conflict?

George and Tom have created conflicting changes (changesets 101 and 103) and Tom has shared his with the master. If George were to incorporate from the master, he would have to resolve the conflict. What if Tom had made a set of follow-up changes that he has not yet shared? If these changes resolve the conflict, then it is likely better for George to wait for Tom to share his new changes. Tom's sharing action would be the best way to resolve George's TEXTUAL \times relationship 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 \checkmark 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 [10]. 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.

5.2 Exhaustively enumerating the space

This section exhaustively enumerates the space described in Section 5.1. Section 5.2.1 describes all possible repository topologies among three developers. Section 5.2.2 describes all possible situations that can arise during collaborative development in those topologies.

5.2.1 Repository topologies

A repository topology describes which repositories may share changes with, and incorporate changes from, which other repositories. For example, centralized VCSes generally restrict each developer to share with and incorporate from only a single "master" repository. In practice, there are many ways to restrict collaborative development to conform to topologies, including VCS constraints, corporate guidelines, and developer practices. Distributed VCSes generally allow unrestricted sharing and incorporating. However, in practice, unrestricted topologies are rare. The centralized use case, with a "master" repository, is the most common use case even for distributed approaches [44].

We call repository *A* the *child* of repository *B* (and *B* the *parent* of *A*) if *A* can share changes with and incorporate changes from *B*. Note that *A* can be *B*'s child and parent at the same time.

We considered all possible repository relationships (Section 3.2) between two developers *A* and *B* and applied all permitted operations. We represented all other repositories with a single repository *C*. For three repositories *A*, *B*, and *C*, considering all possible parent / child relationships yields $2^6 = 64$ distinct topologies. After exhaustively considering those 64 topologies (see Figure 12 in Section 10), we found that there are three classes of topologies that are relevant to consider from *A*'s point of view. These are:

- *A* and *B* share a common parent ($\mathbb{T}1$ in Figure 12),
- *A* is *B*'s child ($\mathbb{T}7$ in Figure 12), and
- *A* is *B*'s non-child descendant ($\mathbb{T}3$ in Figure 12).

The other classes are not relevant because they do not allow *A* to perform actions, do not allow *A* and *B* to collaborate, or are combinations of other classes (adding no new interesting information). See Section 10 and Figure 12 for more information.

Figure 8 shows the three relevant classes of topologies. These three classes represent all the relevant, distinct topologies, in term of *A*'s abilities. This classification focuses on what *A* can do independently of asynchronous actions by other repositories. For example, in a $\mathbb{T}7$ topology, *A* may be able to share changesets with *B*, but if *B* incorporates them first, *A* loses that ability. Figure 8 differs from Figure 12 in that Figure 8 includes all other possible repositories, whereas Figure 12 included only those via which information might flow between *A* and *B*.

5.2.2 Collaborative situations

The repository topologies restrict what the developers may and may not do at any given point during development. The relationships (recall Section 3.2) between the repositories result in further restrictions.

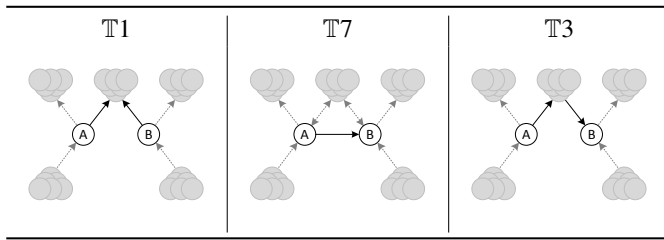


Fig. 8. The three relevant classes of topologies that are distinct in term of *A*'s abilities. An edge from repository *X* to repository *Y* indicates that *X* is *Y*'s child. The solid, black edges must be present for a topology to belong to a given class, whereas the dotted, gray edges may or may not be present. The gray groupings of nodes represent arbitrary, connected structures.

At a moment when new development is not taking place, a goal of collaborating developers is to incorporate everyone's changes together, reaching **SAME** relationships. Even during development, reaching a mutually consistent single state is a goal (to the extent that is possible). Thus, for each topology, we considered *A*'s relationships with *B* and with *C*, and answered the following questions to identify what actions the developers may wish to perform:

- \mathcal{A} Can *A* share?¹
- \mathcal{B} Does sharing **reduce** the number of future actions *A* will have to perform to become **SAME** with *B*?
- \mathcal{C} Can *A* incorporate?
- \mathcal{D} Does incorporating **reduce** the number of future actions *A* will have to perform to become **SAME** with *B*?
- \mathcal{E} Do new changesets in *C* **increase** the number of actions *A* has to perform to become **SAME** with *B*?

By exhaustively examining all these situations, which actions *A* can perform, and which of those help *A* become closer to *B*, we identified the kinds of guidance (recall Section 5.1.3) that help developers make VC decisions. For example, for $\mathbb{T}1$ and $\mathbb{T}3$ topologies (see Figure 8), if *A* is **AHEAD** of *B* and **SAME** with its parent, *A* can neither share nor incorporate, which means *A cannot* perform an action to become **SAME** with *B*, and *B must* be the one to act.

Figure 9 answers the above five questions for the $\mathbb{T}1$ topology. For example, if *A*, *B*, and *C* are all **SAME** (top row in Figure 9), then *A* can neither share (\mathcal{A}) nor incorporate (\mathcal{C}) changes, and since *A* can do neither, it does not make sense to ask whether sharing (\mathcal{B}) and incorporating (\mathcal{D}) reduces the number of actions *A* has to perform to become **SAME** with *B*. Finally, since *A* and *B* are already **SAME**, new changesets in *C* do not affect that relationship (\mathcal{E}).

By contrast, if *A* is **BEHIND** *B* and **AHEAD** of *C* (9th row in Figure 9), *A* can share (\mathcal{A}) — the $\mathbb{T}1$ topology only allows *A* to share with *C* — and sharing does not reduce the number of actions *A* has to perform to become **SAME** with *B* (\mathcal{B}), because *B* would still need to share changes with *C*, and then *A* would still need to incorporate *B*'s changes from *C*. *A* cannot incorporate (\mathcal{C}) — the $\mathbb{T}1$ topology only allows *A* to incorporate from *C* — and thus it does not make sense to ask whether incorporating (\mathcal{D}) reduces the number of actions *A* has to perform to become

<i>A</i> and <i>B</i>	<i>A</i> and <i>C</i>	\mathcal{A}	\mathcal{B}	\mathcal{C}	\mathcal{D}	\mathcal{E}
SAME	SAME	no	—	no	—	no
	AHEAD	impossible				
	BEHIND	no	—	yes	no	no
	all others	impossible				
AHEAD	SAME	no	—	no	—	yes
	AHEAD	yes	yes	no	—	yes
	all others	no	—	yes	yes	no
BEHIND	SAME	no	—	no	—	yes
	AHEAD	yes	no	no	—	yes
	all others	no	—	yes	no	no
all others	SAME	no	—	no	—	yes
	AHEAD	yes	no	no	—	yes
	all others	no	—	yes	no	no

Fig. 9. Answers to the five questions \mathcal{A} – \mathcal{E} for a $\mathbb{T}1$ topology.

SAME with *B*. Finally, new changes in *C* increase the number of actions *A* has to perform to become **SAME** with *B* because *B* would first have to incorporate those new changes before sharing its changes with *C* (\mathcal{E}).

The information in Figure 9 that describes the $\mathbb{T}1$ topology is a superset of the actions available to the developers in the other topologies.

We used these questions to identify the five types of guidance information from Section 5.1.3. Those five types describe completely the answers to these questions. As a result, making the developers aware of the guidance informs their decisions in avoiding and resolving conflicts.

5.3 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 two situations from Volde-mort, one resulting in **BUILD** \times and one in **TEST** \times , for which the conflicts could have been detected earlier and, potentially, never committed to the repository. The information available to developers for these kinds of conflicts is similar to that for **TEXTUAL** \times .

5.3.1 BUILD \times conflict due to 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`. Eight minutes later, the developer merged in another branch (`f68e3`), which resolved the compilation problem.

In this case, a tool could have speculatively told the developer about the compilation error that would arise as a 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.

¹In a DVCS, push shares changes and pull incorporates them.

5.3.2 TESTX conflict due to 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 difference between these two branches was very large (63,413 lines), Git successfully merged these changesets. Test `voldemort.store.http.HttpStoreTest::test-BadPort()` did not fail in 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.

5.4 Relating CVCS and DVCS

This paper has focused on DVCS. However, proactive detection of collaboration conflicts is similarly applicable to CVCS. There are two ways in which this work can be extended to CVCS.

First, from the point of view of collaborative information, DVCS repositories are equivalent to branches in both DVCS and CVCS. In CVCS, developers use separate branches where DVCS developers use either distributed repositories or branches. In that case, the branches can be classified into exactly the seven topology classes from Section 5.2.1. In fact, Microsoft Beacon (a version of Crystal that Microsoft built jointly with us) proactively detects conflicts in exactly this way. Beacon works with a CVCS used internally by Microsoft. Case studies of Beacon are future work.

Second, while the technique we have described considers changes only once they are committed into changesets, it can be extended to consider changes as soon as the developer makes them, or as soon as the developer saves the source files. Considering such changes may reduce further the time before conflicts are detected, which in turn may reduce the frequency and duration of conflicts. On the other hand, considering such changes could also detect conflicts among temporary changes that the developer does not intend to share with others.

Fundamentally, the same information is available in collaborative development that uses each of CVCS and DVCS, at the time of development, and our technique applies to both. (Note, however, that different information is stored by DVCS and CVCS histories, so the retrospective analysis from Section 4 we performed on DVCS repositories could not have been performed on CVCS repositories.)

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?

Our tool, Crystal, conveys the key information without overwhelming or distracting the developer, in three ways.

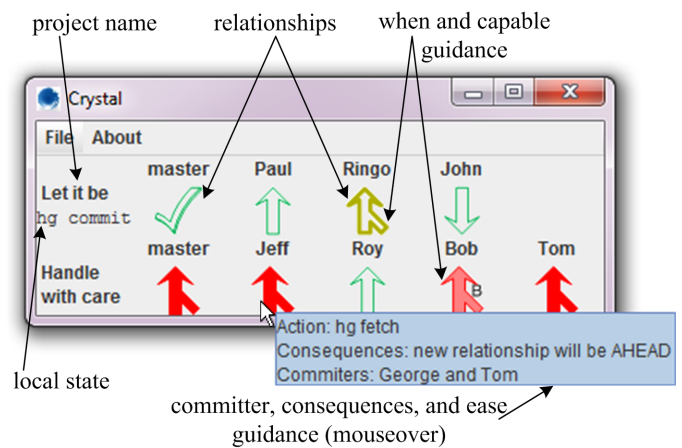


Fig. 10. A screenshot of George’s view of Crystal. George is following two projects under development: “Let it be” and “Handle with care”. The former has four observed collaborators: George, Paul, Ringo, and John; the latter has five: George, Jeff, Roy, Bob, Tom. Crystal shows George’s local state and his relationships with the master repository and the other collaborators, as well as guidance based on that information.

First, a taskbar icon in the system tray reports the most severe state for all tracked repositories. A developer who prefers to receive limited but critical information need never open the main window. (Crystal never opens any window asynchronously.)

Second, the main window compactly *summarizes* all projects and relationships, allowing a developer to instantly scan it to identify situations that may require attention. The main window displays icons exploiting color and shape redundantly and in stable locations (rather than, say, a textual list that a developer would have to read and interpret). Each icon’s fixed color represents the severity of the situation.

Third, full, detailed information about each relationship, action, and guidance is *available but hidden* until a developer shows specific interest in it. When the developer mouses over an icon, a tooltip displays all the information discussed in Section 5.

Crystal works with the Mercurial and Git DVCSes. 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 10 shows a screenshot of Crystal’s main window. 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. The window displays a row of icons (see Figure 11) for each of a developer’s projects.



Fig. 11. Crystal associates an icon with each of the seven relationships. Each icon has a fixed color, which represents the severity of the relationship (Section 3.2): relationships that require no merging are green, those that can be merged automatically are yellow, and those that require manual merging are red. Icons can be shaded to indicate a developer's capability to affect this relationship at this time. Solid, unsaturated, and hollow icons indicate the developer can, might, and cannot affect this relationship, respectively. Here, the icons are shown solid; Figure 10 includes examples of unsaturated and hollow icons.

When and *Capable* guidance is represented by the shading of the icon. If George has the ability to affect a relationship now, the icon is solid. 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. As shown in Figure 10, 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 a resolve operation 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.

Our initial, anecdotal experience has suggested that Crystal affects developers' workflow by (1) prompting communication between developers who create a conflict, and (2) reminding developers to incorporate changes before beginning work. Further, Crystal affects managers' workflow by increasing their awareness of which developers have made, shared, and incorporated changes.

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 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.

6.3 Precision and timeliness in conflict detection

Crystal precisely reports actual conflicts, determining the relationship between two developers' states by actually creating the merged artifact. In other words, to find out what would happen if George and Ringo merged their code, Crystal does the merge in the background: it makes a copy of George's code and incorporates Ringo's changes. Similarly, once Crystal creates the merged code artifact, it attempts to compile and to execute the test suite on that artifact. Again, Crystal only reports a compilation or testing conflict when the build or a test actually fails. Because the computation happens in the background, the developers can continue to work without interruption. In certain situations, we expect the developers to ignore Crystal, much as they sometimes ignore project bulletin boards and email.

By creating the merged artifact in the background, Crystal uses *speculative analysis* [7] to detect conflicts. In contrast, awareness tools [4], [15], [23] notify developers when they *might* have conflicting changes. This approximation is computed differently in various tools. Some determine if a co-developer is working in the same file, some report any change to the repository (e.g., FASTDash [4]), others report concurrent changes to the AST (e.g., Syde [23]), etc. These approaches can lead to the inclusion of false positives — reporting potential conflicts that do not evolve into actual conflicts. Furthermore, few current awareness tools try to automatically detect higher-order merge conflicts; by contrast, Crystal is precise as it uses the project's tool chain to dynamically detect conflicts by execution of the build system and test suites. We discuss further differences between Crystal and awareness tools in Section 7.

Crystal can, in rare situations, report conflicts about which the developers may not wish to know. Changesets that are later *discarded* can cause a teammate to see a pending conflict that later disappears. This can happen when a developer commits exploratory code, a partial change, or a change that is later determined to be undesirable.

6.4 Scalability

This section describes how Crystal's design allows it to scale to large projects and compute the relevant information efficiently.

6.4.1 Large projects

Crystal scales to large projects that involve many developers and repositories. A developer explicitly instructs Crystal (via a GUI or a configuration file) which repositories to observe. For example, a developer may be interested in only the relationships with other developers in his collaborative team and the per-team 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. In particular, the repositories being monitored need not be using Crystal—they only have to be accessible to Crystal (see Section 6.4.2 below).

For detecting test failures, Crystal allows a developer to select any subset of the tests to execute. 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.

While the current implementation of Crystal runs on the developer's machine, its architecture can be extended to offload the computation onto a central server (perhaps an integration server), a cluster of machines, or the cloud. Projects whose build and test scripts cannot feasibly run on the developer's machine in a reasonable amount of time may require such offloading.

6.4.2 Computation efficiency

Crystal provides a developer with information on his development state and the relationships between his repository and collaborators' repositories. Thus, Crystal needs access to that developer's repository and working copy (if the working copy is inaccessible or nonexistent, Crystal does not report certain local states, e.g., uncommitted), and the locations of the other repositories. In some development environments, access to others' repositories is trivial. For example, many corporate development configurations include a common file system. In other environments, it is possible for developers to have their local repositories on machines that are often offline. In such an environment, each developer (even those not using Crystal) must make his repository available to other developers. One simple approach is to symbolically link their repository to a Dropbox² shared folder.

To limit the computation necessary to extract the relationships between repositories, Crystal follows the following algorithm. First, Crystal checks the history of the two repositories to identify the changesets each contains, and only re-computes the relationship if at least one history has changed. If the sets of changesets are the same, then the relationship is **SAME**. If one repository contains strictly more (respectively fewer) changesets, it is **AHEAD** (respectively **BEHIND**). If both repositories contain changesets the other does not (and Crystal has not previously computed their relationship), Crystal makes a local clone of one repository and uses the VCS to attempt to incorporate the changesets from the other repository. If the VCS reports a problem with incorporation, the relationship is **TEXTUAL✗**. If the integration succeeds, Crystal runs the build script. If that script fails, the relationship is **BUILD✗**. Finally, if the build script succeeds, Crystal runs the test suite and determines whether the relationship is **TEST✗** or **TEST✓**.

Cloning repositories, especially remote ones, can be costly. To address this issue (and to enable faster start-up times), Crystal keeps a cached clone of each project, bringing it up to date before updating the relevant relationship. This has significantly increased Crystal's performance in all common cases. In the rare and discouraged situation of changing existing VC history (e.g., rebasing), the cache may contain changesets that no longer exist in a repository. This can cause problems and require the developer to clear the cache.

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.

²<http://www.dropbox.com>

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 [32]. Developers eschew parallel work to avoid having to resolve conflicts when committing changes [19], or rush their work into the trunk to avoid being the developer who would have to resolve conflicts [13]. In practice, anti-patterns for parallel software development emerge that hinder collaboration [2], [6]. 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 [17].

Several observational and laboratory experiments empirically demonstrate that collaborative awareness benefits configuration management by reducing use of shared resources, increasing project-related communication, and detecting some conflicts at the time they are created [4], [15], [42]. 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.

In practice, some branches can cause delays in integration, in part because branches typically have to meet certain criteria before integration [2], [6]. These delays can, in turn, lead to conflicts. Classifying branches based on their integration history can, in theory, reduce delays, and perhaps conflicts [6].

Sarma provides a comprehensive classification of collaborative tools for software development [39]. 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 [41], [40], [1] shows which developers are changing which artifacts (e.g., files) 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” [41, p. 444]. FASTDash [4] 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 [23] reduces false positives via a fine-grained analysis of abstract syntax trees (AST) modifications. Two potentially-conflicting changes to the same file are flagged for a developer only when they also change 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 the users 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 [e.g., method, class, or file] that has a dependency on another program element that has been edited but not checked-in by another developer” [15]. Safe-commit [45] 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.

Instead of considering the conflicts that arise when integrating the code of two developers, it is possible to consider integrating the code of all developers working on a project at once. Performing this analysis continuously can help discover merge problems early [22].

Our approach suffers fewer false positives and fewer false negatives than previous awareness approaches [8] for four 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 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 Version Control Systems

Rochkind introduced the first source code control system in 1975 [33]. Since then, numerous similar systems — characterized by a centralized shared repository — have been developed and deployed, including RCS [43], CVS [20], and Subversion [11]. More recently, a set of DVCSes have emerged, including Bazaar, Mercurial, and Git. These systems do not rely on a centralized repository, are less dependent on network availability, and allow more freedom to the collaborators in terms of branching, merging, and keeping multiple repositories. As we have discussed in Section 5.4, neither the distinctions between centralized and distributed version control, nor those among specific VCSes [12], [28], [31] prevent the technique we have presented from proactively detecting conflicts. The only exception to this might be that DVCSes may encourage more frequent branching and merging [44], which likely provides additional opportunities for our technique. Perry et al. [32] empirically document these variations, and consequences of these variations with respect to quality and schedule, in how software teams perform work in parallel.

7.4 Mining software repositories

Ball et al. [3] 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 [48], to identify common error patterns [27], to predict component failures [30], 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.

7.5 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 [24], [26]. 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 [36], [37], [38], [18] applies the same idea to testing: it uses excess cycles on a developer’s workstation to continuously run regression tests in the background. It reduces the time and energy required to keep code well-tested and prevents 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 requires small unit tests that can execute quickly. Test factoring automatically carves large system-wide tests into such unit tests [16], [35].

Recent work has investigated real-time integration to decrease developers’ hesitation in committing changes using centralized version control [21]. Like FASTDash, this approach aims to help developers avoid conflicts. In contrast to FASTDash (but similarly to Crystal), it computes rather than predicts the presence of merge conflicts.

Similarly to continuous compilation, execution, testing, and integration, our approach is reactive to certain developer actions — committing changes — and proactive with respect to others — sharing and incorporating those changes. Unlike these other approaches, our approach focuses on detection of collaboration conflicts among developers.

8 THREATS TO VALIDITY

This section discusses threats to the validity of our results.

Construct: The version control histories tell us when a TEXTUAL \times or TEXTUAL \checkmark 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?

DVCS histories only contain information about incorporate operations from the TEXTUAL \checkmark and TEXTUAL \times states; nothing is recorded when a developer incorporates from the BEHIND state or pushes from the AHEAD state. DVCS histories do not record when share operations take place in BEHIND and AHEAD states. (DVCS disallows sharing in TEXTUAL \checkmark and TEXTUAL \times states, although a special flag allows such sharing, which is then also omitted from the record.)

Our analysis used the projects’ test suites to detect test failures. A merge might cause other semantic errors that were not detected by the tests.

Internal: Our experiments (see Section 4) are in the context of DVCSes, which differ from CVCSes [12], [28], [31]. The complete effects of the VCS on developer behavior is not established [5], [34], [46] If DVCSes encourage more frequent branching and merging [44], 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. Our largely similar results to Zimmermann [47] 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






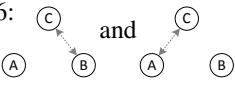



	Description	A's responsibilities	What affects A's relationships and abilities
T1: 	A and B are C's children and using C is the only way for A and B to communicate changesets.	Sharing changesets created at A with C.	Only changesets in A and in B affect A's relationship with B. Changesets in A, in B, and in C affect which actions A can perform to affect its relationship with B.
T2: 	A and B are C's parents and there is no other way for changes to be communicated between A and B.	A cannot perform any actions.	C is responsible for incorporating and sharing all new changesets.
T3: 	A is B's descendant but not child.	Sharing changesets created at A with C.	Only changesets in A and in B affect A's relationship with B. Changesets in A, in B, and in C affect which actions A can perform to affect its relationship with B.
T4: 	A is B's ancestor but not parent.	A cannot perform any actions.	B and C are responsible for incorporating and sharing all new changesets.
T5: 	Five configurations with either 3 or 4 of the gray, dotted edges in the diagram.	The edges of each of these topologies can be represented as the union of the edges of topologies from T1–T4. A has the same responsibilities and abilities as those topologies the union of whose edges make up this topology.	
T6: 	Seven configurations with 0, 1, or 2 of the gray, dotted edges in the diagrams.	A and B are detached and cannot collaborate.	
T7: 	Sixteen configurations with 0 or more of the gray, dotted edges in the diagram. A is B's child. A's capabilities with regard to its relationship with B are neither affected by the edges between A and C nor between B and C.	Sharing changesets created at A with B.	Only changesets in A and in B affect A's relationship with B. Changesets in A, in B, and in C affect which actions A can perform to affect its relationship with B.
T8: 	Sixteen configurations with 0 or more of the gray, dotted edges in the diagram. A is B's parent. A's capabilities with regard to its relationship with B are affected by neither the edges between A and C nor those between B and C.	Because A cannot perform any actions directly with B, these topologies either do not allow A to perform any actions at all or are combinations of T1–T4.	
T9: 	Sixteen configurations with 0 or more of the gray, dotted edges in the diagram. A is B's child and parent. A's capabilities with regard to its relationship with B are neither affected by the edges between A nor C and between B and C.	Because A being B's parent does not allow A to perform any actions, these topologies are identical to T7, from A's point of view.	

Fig. 12. The 64 topologies over three repositories can be grouped into nine classes. Two topologies within the same class are identical in terms of the actions A and B can perform.

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 in practice, 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 or abandons 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, indicates that (1) conflicts are the norm rather than the exception, (2) 16% of all merges required human effort to resolve textual conflicts, (3) 33% of merges that were reported to contain no textual conflicts by the VCS in fact contained higher-order conflicts, and (4) conflicts persist, on average, for 3.2 days (with a median conflict persisting 0.7 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 [47]. We expand on his work (1) by comparing actual merges from project histories to merges that could have taken place successfully earlier than they did, and (2) by 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 allows them to identify and fix conflicts before they fester. This is one useful and practical step in reducing some of the costs and difficulties of collaborative software development.

10 COLLABORATIVE SITUATIONS

This section exhaustively enumerates (as overviewed in Section 5.2) the space described in Section 5.1. It first describes the space of all possible repository topologies among three developers and then all possible situations that can arise during collaborative development in those topologies.

We represent a *repository topology* by a directed graph. A node represents a repository. (We use the node's label to refer to both to the repository and to the developer who owns that repository.) A directed edge from one repository (the *child*) to another repository (the *parent*) represents the ability of (the owner of) the child to share changes with and incorporate changes from the parent. Directed paths connect *descendants* to their *ancestors*. For example, a simple topology ($\mathbb{T}1$) with a single master repository and two children would be represented by three nodes, with an edge from each child to the master. Two nodes may be each

other's parents and children simultaneously, represented by one directed edge in each direction or by a single two-headed edge. Such a relationship means that the owner of either repository may share changes with and incorporate changes from the other repository.

To determine the information specific to VC operations that can help make better collaborative decisions, given a topology between two developers, A and B , we consider all possible repository relationships (Section 3.2) and apply all operations permitted by the topology. Without loss of generality, we consider only A 's point of view (because A and B are symmetric). We first represent all other repositories with which A and B may interact as single repository C . For three nodes (A , B , and C), there are six potential directed edges: $A \rightarrow B$, $B \rightarrow A$, $A \rightarrow C$, $C \rightarrow A$, $B \rightarrow C$, $C \rightarrow B$. Therefore, there are $2^6 = 64$ potential topologies with three nodes. Figure 12 enumerates these 64 topologies and groups them into 9 classes, named $\mathbb{T}1$ – $\mathbb{T}9$, that are distinct from the global point of view of the actions they allow developers to perform. In Figure 8, we considered how these nine classes differ from A 's point of view, and further grouped these nine classes into three classes from that perspective.

Three of these topology classes, $\mathbb{T}1$, $\mathbb{T}3$, and $\mathbb{T}7$, have the following three properties: each topology (1) allows A to perform actions, (2) allows A to collaborate with B , and (3) is not a combination of other classes (if it were, it would provide no new information about A 's capabilities). We call these three classes relevant, from A 's point of view (recall Section 5.2.1).

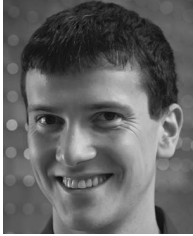
ACKNOWLEDGMENTS

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] 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," *Proc. Int'l Workshop Cooperative and Human Aspects of Software Eng.*, pp. 1-4, May 2008.
- [2] B. Appleton, S.P. Berczuk, R. Cabrera, and R. Orenstein, "Streamed Lines: Branching Patterns for Parallel Software Development," *Proc. Pattern Languages of Programs Conf.*, 1998.
- [3] T. Ball, J.-M. Kim, A.A. Porter, and H.P. Siy, "If Your Version Control System Could Talk," *Proc. Workshop Process Modelling and Empirical Studies of Software Eng.*, May 1997.
- [4] J.T. Biehl, M. Czerwinski, G. Smith, and G.G. Robertson, "FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams," *Proc. SIGCHI Conf. Human Factors in Computing Systems*, pp. 1313-1322, Apr. 2007.
- [5] C. Bird, P.C. Rigby, E.T. Barr, D.J. Hamilton, D.M. Germán, and P.T. Devanbu, "The Promises and Perils of Mining Git," *Proc. Sixth IEEE Int'l Working Conf. Mining Software Repositories*, pp. 1-10, 2009.
- [6] C. Bird and T. Zimmermann, "Assessing the Value of Branches with What-If Analysis," *Proc. ACM SIGSOFT 20th Int'l Symp. Foundations of Software Eng.*, 2012.

- [7] Y. Brun, R. Holmes, M.D. Ernst, and D. Notkin, "Speculative Analysis: Exploring Future States of Software," *Proc. FSE/SDP Workshop Future of Software Eng. Research*, pp. 59-63, Nov. 2010.
- [8] Y. Brun, R. Holmes, M.D. Ernst, and D. Notkin, "Crystal: Precise and Unobtrusive Conflict Warnings," *Proc. 19th ACM SIGSOFT Symp. and 13th European Conf. Foundations of Software Eng.*, Sept. 2011.
- [9] Y. Brun, R. Holmes, M.D. Ernst, and D. Notkin, "Proactive Detection of Collaboration Conflicts," *Proc. 19th ACM SIGSOFT Symp. and 13th European Conf. Foundations of Software Eng.*, pp. 168-178, Sept. 2011.
- [10] M. Cataldo, P.A. Wagstrom, J.D. Herbsleb, and K.M. Carley, "Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools," *Proc. 20th Anniversary Conf. Computer Supported Cooperative Work*, pp. 353-362, Nov. 2006.
- [11] B. Collins-Sussman, "The Subversion Project: Building a Better CVS," *Linux*, vol. 3, no. 94, 2002.
- [12] R. Conradi and B. Westfechtel, "Version Models for Software Configuration Management," *ACM Computing Surveys*, vol. 30, no. 2, pp. 232-282, 1998.
- [13] C.R.B. de Souza, D. Redmiles, and P. Dourish, "'Breaking the Code,' Moving between Private and Public Work in Collaborative Software Development," *Proc. Int'l ACM SIGGROUP Conf. Supporting Group Work*, pp. 105-114, Nov. 2003.
- [14] P. Dewan, "Dimensions of Tools for Detecting Software Conflicts," *Proc. Int'l Workshop Recommendation Systems for Software Eng.*, pp. 21-25, Nov. 2008.
- [15] P. Dewan and R. Hegde, "Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development," *Proc. European Computer Supported Cooperative Workshop*, pp. 159-178, Sept. 2007.
- [16] S. Elbaum, H.N. Chin, M.B. Dwyer, and J. Dokulil, "Carving Differential Unit Test Cases from System Test Cases," *Proc. 14th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.*, pp. 253-264, 2006.
- [17] J. Estublier and S. Garcia, "Process Model and Awareness in SCM," *Proc. 12th Int'l Workshop Software Configuration Management*, pp. 59-74, Sept. 2005.
- [18] D.S. Glasser, "Test Factoring with Amock: Generating Readable Unit Tests from System Tests," master's thesis, MIT Dept. of EECS, Aug. 2007.
- [19] R.E. Grinter, "Using a Configuration Management Tool to Coordinate Software Development," *Proc. Conf. Organizational Computing Systems*, pp. 168-177, Aug. 1995.
- [20] D. Grune, "Concurrent Versions System, a Method for Independent Cooperation," Technical Report IR 113, Vrije Universiteit, 1986.
- [21] M.L. Guimarães and A. Rito-Silva, "Towards Real-Time Integration," *Proc. ICSE Workshop Cooperative and Human Aspects of Software Eng.*, pp. 56-63, May 2010.
- [22] M.L. Guimarães and A.R. Silva, "Improving Early Detection of Software Merge Conflicts," *Proc. Int'l Conf. Software Eng.*, 2012.
- [23] L. Hattori and M. Lanza, "Syde: A Tool for Collaborative Software Development," *Proc. ACM/IEEE 32nd Int'l Conf. Software Eng.*, pp. 235-238, May 2010.
- [24] P. Henderson and M. Weiser, "Continuous Execution: The VisiProg Environment," *Proc. Eighth Int'l Conf. Software Eng.*, pp. 68-74, Aug. 1985.
- [25] S. Horwitz, J. Prins, and T. Reps, "Integrating Noninterfering Versions of Programs," *ACM Trans. Programming Languages and Systems*, vol. 11, pp. 345-387, July 1989.
- [26] R.R. Karinthi and M. Weiser, "Incremental Re-Execution of Programs," *Proc. Symp. Interpreters and Interpretive Techniques*, pp. 38-44, June 1987.
- [27] B. Livshits and T. Zimmermann, "DynaMine: Finding Common Error Patterns by Mining Software Revision Histories," *Proc. 10th European Software Eng. Conf. Held Jointly with 13th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.*, pp. 296-305, Sept. 2005.
- [28] T. Mens, "A State-of-the-Art Survey on Software Merging," *IEEE Tran. Software Eng.*, vol. 28, no. 5, pp. 449-462, May. 2002.
- [29] K. Muşlu, Y. Brun, R. Holmes, M.D. Ernst, and D. Notkin, "Speculative Analysis of Integrated Development Environment Recommendations," *Proc. ACM Int'l Conf. Object Oriented Programming Systems Languages and Applications*, Oct. 2012.
- [30] N. Nagappan, T. Ball, and A. Zeller, "Mining Metrics to Predict Component Failures," *Proc. 28th Int'l Conf. Software Eng.*, pp. 452-461, 2006.
- [31] B. O'Sullivan, "Making Sense of Revision-Control Systems," *Queue*, vol. 7, no. 7, pp. 30-40, 2009.
- [32] D.E. Perry, H.P. Siy, and L.G. Votta, "Parallel Changes in Large-Scale Software Development: An Observational Case Study," *ACM Trans Software Eng. and Methodology*, vol. 10, pp. 308-337, July 2001.
- [33] M.J. Rochkind, "The Source Code Control System," *IEEE Trans. Software Eng.*, vol. 1, no. 4, pp. 364-370, Dec. 1975.
- [34] C. Rodriguez-Bustos and J. Aponte, "How Distributed Version Control Systems Impact Open Source Software Projects," *Proc. Ninth IEEE Working Conf. Mining Software Repositories*, pp. 36-39, 2012.
- [35] D. Saff, S. Artzi, J.H. Perkins, and M.D. Ernst, "Automatic Test Factoring for Java," *Proc. IEEE/ACM 20th Int'l Conf. Automated Software Eng.*, pp. 114-123, Nov. 2005.
- [36] D. Saff and M.D. Ernst, "Reducing Wasted Development Time via Continuous Testing," *Proc. 14th Int'l Symp. Software Reliability Eng.*, pp. 281-292, Nov. 2003.
- [37] D. Saff and M.D. Ernst, "Continuous Testing in Eclipse," *Proc. Second Eclipse Technology Exchange Workshop*, Mar. 2004.
- [38] D. Saff and M.D. Ernst, "An Experimental Evaluation of Continuous Testing during Development," *Proc. ACM SIGSOFT Int'l Symp. Software Testing and Analysis*, pp. 76-85, July 2004.
- [39] A. Sarma, "A Survey of Collaborative Tools in Software Development," Technical Report UCI-ISR-05-3, Univ. of California, Irvine, Inst. of Software Research, 2005.
- [40] A. Sarma, G. Bortis, and A. van der Hoek, "Towards Supporting Awareness of Indirect Conflicts Across Software Configuration Management Workspaces," *Proc. 22nd IEEE/ACM Int'l Conf. Automated Software Eng.*, pp. 94-103, Nov. 2007.
- [41] A. Sarma, Z. Noroozi, and A. van der Hoek, "Palantir: Raising Awareness among Configuration Management Workspaces," *Proc. 25th Int'l Conf. Software Eng.*, pp. 444-454, May 2003.
- [42] A. Sarma, D. Redmiles, and A. van der Hoek, "Empirical Evidence of the Benefits of Workspace Awareness in Software Configuration Management," *Proc. 16th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.*, pp. 113-123, Nov. 2008.
- [43] W.F. Tichy and W.F. Tichy, "RCS—A System for Version Control," *Software: Practice and Experience*, vol. 15, pp. 637-654, 1985.
- [44] C. Walrad and D. Strom, "The Importance of Branching Models in SCM," *Computer*, vol. 35, no. 9, pp. 31-38, 2002.
- [45] J. Wloka, B. Ryder, F. Tip, and X. Ren, "Safe-Commit Analysis to Facilitate Team Software Development," *Proc. Int'l Conf. Software Eng.*, pp. 507-517, May 2009.
- [46] J. Wuttke, I. Beschastnikh, and Y. Brun, "Effects of Centralized and Distributed Version Control on Commit Granularity," *Tiny Trans. Computer Science*, Sept. 2012.
- [47] T. Zimmermann, "Mining Workspace Updates in CVS," *Proc. Fourth Int'l Workshop Mining Software Repositories*, May 2007.
- [48] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," *Proc. 26th Int'l Conf. Software Eng.*, pp. 563-572, 2004.



Yuriy Brun received the MEng degree from the Massachusetts Institute of Technology in 2003 and the PhD degree from the University of Southern California in 2008. He completed his postdoctoral work in 2012 at the University of Washington as a CI Fellow. He is an assistant professor in the School of Computer Science at the University of Massachusetts, Amherst. His research interests include software engineering, distributed systems, and self-adaptation. He is a

member of the IEEE, the ACM, and ACM SIGSOFT. More information is available on his homepage: <http://www.cs.umass.edu/brun/>.



Reid Holmes completed an NSERC postdoctoral fellowship at the University of Washington in 2010 following receiving the PhD degree from the University of Calgary in 2008. He is an assistant professor in the Cheriton School of Computer Science at the University of Waterloo. His research interests include understanding how software engineers build and maintain complex systems; this understanding is generally translated into tools and techniques that can be

validated in practice. His prior research focused on pragmatic software reuse and source code recommendation systems. More information is available on his homepage: <https://cs.uwaterloo.ca/rtholmes/>.



Michael D. Ernst is an associate professor in Computer Science & Engineering at the University of Washington. He was previously a tenured professor at MIT, and before that a researcher at Microsoft Research. His research aims to make software more reliable, more secure, and easier (and more fun!) to produce. His primary technical interests include software engineering and related areas, including programming languages, type theory, security,

program analysis, bug prediction, testing, and verification. His research combines strong theoretical foundations with realistic experimentation, with an eye to changing the way that software developers work. More information is available on his homepage: <http://homes.cs.washington.edu/mernst/>.



David Notkin (1955-2013) received the ScB degree from Brown University in 1977 and the PhD degree from Carnegie Mellon University in 1984. He served as a professor and Bradley chair Computer Science & Engineering at the University of Washington, which he joined in 1984. His research interests include software engineering in general and in software evolution in particular. He received the US National Science Foundation Presidential Young Investi-

gator Award, served as the program chair of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering, served as program cochair of the 1995 International Conference on Software Engineering, chaired the steering committee of the International Conference on Software Engineering, served as the general chair of the 2013 International Conference on Software Engineering, served as a charter associate editor and later as editor-in-chief of the *ACM Transactions on Software Engineering and Methodology*, served as an associate editor of the *IEEE Transactions on Software Engineering*, received the ACM SIGSOFT Distinguished Service Award, the ACM SIGSOFT Outstanding Research Award, the ACM SIGSOFT Influential Educator Award, and the A. Nico Habermann Award, served as the chair of ACM SIGSOFT, served as the department chair of computer science and engineering, and received the University of Washington Distinguished Graduate Mentor Award. He is a fellow of the IEEE and ACM.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.