Predicting Development Trajectories to Prevent Collaboration Conflicts

Yuriy Brun[†], **Kıvanç Muşlu**[†], **Reid Holmes**[✱], **Michael D. Ernst**[†], **David Notkin**[†]

[†]Computer Science & Engineering
University of Washington
Seattle, WA, USA
{brun, kivanc, mernst, notkin}@cs.washington.edu

[✱]School of Computer Science
University of Waterloo
Waterloo, ON, Canada
rtholmes@cs.uwaterloo.ca

ABSTRACT

The benefits of collaborative development are reduced by the cost of resolving conflicts. We posit that reducing the time between when developers introduce and learn about conflicts reduces this cost. We outline the state-of-the-practice of managing and resolving conflicts and describe how it can be improved by available state-of-the-art tools. Then, we describe our vision for future tools that can predict likely conflicts before they are even created, warning developers and allowing them to avoid potentially costly situations.

Author Keywords: collaborate development; collaborative conflicts; conflict prediction; conflict detection

ACM Classification Keywords: D.2.3 [Software Engineering]: Coding Tools and Techniques; D.2.6 [Software Engineering]: Programming Environments

General Terms: Design; Human Factors

COSTS OF COLLABORATIVE DEVELOPMENT

Collaborative software development enables work to be parallelized and completed faster. The benefits of collaboration are offset in part by conflicts that may arise when multiple developers work in parallel. As the amount of parallel work increases, so does the frequency of defects [15]. Resolving conflicts is costly, in part because conflicts are often discovered long after they are created. Delays in learning about conflicts add two costs. First, as more time passes between the introduction of the conflict and its detection, the likelihood increases that the relevant assumptions, artifacts, and changes have faded in the developers' minds. Second, the changes grow in size as time passes, which increases the cost of integration and the likelihood that work must be abandoned or revamped.

In this paper, we discuss sources of delays between conflict introduction and detection in today's practice, outline the state-of-the-art technology that exists to reduce those delays, and describe our vision for how to predict conflicts even before they are introduced, further reducing some of the costs associated with collaborative development.

CSCW'12, February 11–15, 2012, Seattle, Washington, USA.

STATE-OF-THE-PRACTICE OF CONFLICT DETECTION

Today, developers use version control to assist collaborative development. Version control systems allow developers to work in parallel and deal with conflicts systematically. Conflicts can arise whenever developers share inconsistent changes. Developers have two suboptimal options: sharing partial changes or waiting until the assigned tasks are complete. Sharing partial changes can unnecessarily break others' builds and tests, preventing effective parallel work. Meanwhile, waiting until tasks are complete creates a delay between the times when conflicts are introduced and when developers learn about them. Even after changes are shared, other developers may further delay incorporating those changes until their own tasks are complete because incorporating earlier may distract the developers by forcing them to resolve conflicts.

Version control systems are in widespread industrial use and represent today's state-of-the-practice in collaborative development. Since the first source code control system in 1975 [16], numerous similar systems — characterized by a centralized shared repository — have been developed and deployed, including RCS [22], CVS [10], and Subversion [5]. Distributed version control systems have recently emerged, including Bazaar, Code Co-op, Git, and Mercurial [14]. These systems do not rely on a centralized repository, allow more freedom to the collaborators in terms of branching, merging, and keeping multiple repositories, and are less dependent on network availability. Distributed version control systems may encourage more frequent branching and merging, which may create more conflicts but also reduce the cost of their resolution [23].

STATE-OF-THE-ART OF CONFLICT DETECTION

Fear of conflicts can adversely affect developers' behavior. For example, developers sometimes avoid working in parallel to ensure not running into conflicts [9]. Other times, developers may share their code hastily in an attempt to avoid responsibility for expected conflicts [6]. Making accurate information about conflicts, and about lack of conflicts, available can lead to better risk management and a reduction in conflict-related costs [8].

Awareness tools help developers better understand how their changes interact with those of others [17]. These benefits have been verified experimentally [2, 7, 20], suggesting that improving developer awareness can decrease development costs and perhaps reduce defect frequency [15].

Sarma provides a comprehensive classification of collaborative tools for software development [17]. Palantír [19, 18, 1] shows which developers are changing which artifacts, and by how much. FASTDash [2] is an interactive visualization — a spatial representation of which files each developer is editing — that helps developers understand what other team members are doing. Syde [11] increases precision of awareness tools by using a fine-grained analysis of the changed abstract syntax trees (ASTs). Two potentially conflicting changes to the same file are flagged for a developer only when they also affect changes to the same parts of the underlying ASTs. CollabVS "detects a potential conflict when a user starts editing a program element that has a dependency on another program element that has been edited but not checked-in by another developer" [7]. Safe-commit [24] does the deepest program dependence analysis, identifying changes that are guaranteed not to cause tests to fail; this allows developers to share some of their changes earlier. YooHoo [12] can be used to predict build errors early by looking at AST interactions among already created changes that are likely to be merged soon.

Collaborating developers may be distracted when awareness tools suggest or report conflicts inaccurately. Our approach to proactive detection of collaboration conflicts [4] eliminates false positive and false negative reports by using speculative analysis [3] to detect conflicts. Our tool, Crystal[1], creates copies of the developers' code and speculatively merges them in the background using the version control system. Crystal reports textual conflicts if and only if the version control system reports a conflict in a background copy. Further, Crystal builds the merged code and executes its test suite to report when merging changes will result in build and test failures. While awareness tools provide an approximation of what is likely to cause conflicts, Crystal uses the version control system and build and test scripts for conflict detection.

Speculative analysis — the technique behind Crystal — represents the state-of-the-art of conflict detection. As soon as developers create a conflicting change, speculative analysis can notify them about the conflict, reducing the delay between conflict introduction and detection. As we describe in the next section, we believe future tools can do even more to help lower the costs of collaborative development.

OUR VISION: CONFLICT PREDICTION

The need for early conflict *detection* can be reduced — perhaps even eliminated — by conflict *prediction* and *prevention*. We imagine future collaborative development tools that not only observe the changes developers have already made, but also predict the changes developers are likely to make. (We propose below several approaches to making such predictions.) Such tools could analyze the consequences of those changes to predict potential conflicts before developers make the conflicting changes. The tools' interfaces have to be unobtrusive and should not overload the developers while allowing them to manage risk and avoid situations that are likely to cause conflicts. Additionally, developers will know to communicate with others whose future changes are likely to cause conflicts and resolve inconsistencies in their planned changes before they devolve into costly conflicts.

[1]http://crystalvc.googlecode.com

For example, consider a tool that informs a developer that "if you attempt a particular refactoring right now, you will edit lines of code near those another developer is currently editing and likely will need to resolve a conflict manually." Such information can (1) help developers make well-informed decisions about how to proceed, (2) identify risks of performing certain tasks, and (3) encourage communication to prevent costly future conflicts.

Consider a team of collaborative developers, each working in parallel on adding a new, independent feature to a software system. The developers are currently working in separate packages, and their changes cause neither textual, build, nor test conflicts. We envision three ways in which a tool could predict the trajectories of the developers' changes and identify potential future conflicts.

First, for each developer, a tool could consider the changes that developer has already made, and look at the history of this project's development to identify other software artifacts (such as methods, classes, files, tests, configuration parameters, requirements, etc.) that were edited after, or otherwise affected by these changes. The tool could then check for overlaps between artifacts edited by the developers and predict if the changes may cause conflicts. The tool would deliver information to the developers about potential interactions of their possible future changes, such as whether they are likely to edit the same method, or affect the same test or requirement.

Second, a tool could build a model (perhaps using machine learning) of partial changes that have resulted in conflicts in the past and then apply this model to classify the current changes. To do this, the tool would need a history of project development and sets of changes labeled as having or not having resulted in conflicts. Some such sets already exist [4]. For this approach to succeed, machine learning needs to be able to infer, from exiting data, whether changes cause conflicts.

Third, building on the speculative analysis technique used by Crystal, for each developer, a tool could (1) attempt to identify which actions that developer is most likely to perform next, (2) execute sequences of those actions speculatively in the background to create new possible future development states, and then, (3) again speculatively, attempt to merge those states to identify conflicts. To do this, the tool would have to accurately and precisely identify those actions developers are likely to perform. These could be mined from development histories or determined using heuristics, e.g., a developer is more likely to fix complication errors before attempting to refactor or to add a new feature.

While no existing tools implement our vision, some do increase developer awareness of the consequences of future actions. For example, Mylyn [13] shares with others manually-specified information about what tasks the developers will work on, improving awareness and potentially preventing conflicts. However, Mylyn does not detect this information automatically, instead relying on humans to consistently update their lists of tasks. HATARI [21] warns developers when they are working on error-prone artifacts. While not aimed at collaborative development, HATARI has a similar goal to ours:

making developers aware of the risks of their future actions.

While predicting long-term trajectories may be far fetched, we are encouraged by the fact that even short term prediction can immediately bring benefits to the developers. Today, developers are forced to make poorly-informed decisions because they lack access to information about potential conflicts. We envision that future collaborative development tools will make this information available and allow for well-informed decisions.

BIOGRAPHIES

Yuriy Brun is a postdoctoral researcher at the University of Washington. He believes that collaborative development can be significantly aided by improving the understanding of how local changes developers make affect others' changes and the system as a whole.

Kıvanç Muşlu is a PhD student at the University of Washington. He believes that collaborative development can benefit from tools and techniques that capture not only the interactions between the developers and machines but also between the developers themselves.

Reid Holmes is on the faculty of Computer Science at the University of Waterloo. His research interests focus on understanding and improving how developers reason about and evolve software systems.

Michael D. Ernst is on the faculty of Computer Science & Engineering at the University of Washington. His research aims to make software more reliable, more secure, easier, and more fun to produce.

David Notkin is on the faculty of Computer Science & Engineering at the University of Washington, with research and educational interests in software engineering in general and software evolution in particular.

REFERENCES

1. Al-Ani, B., Trainer, E., Ripley, R., Sarma, A., van der Hoek, A., and Redmiles, D. Continuous coordination within the context of cooperative and human aspects of software engineering. In *CHASE* (May 2008), 1–4.
2. Biehl, J. T., Czerwinski, M., Smith, G., and Robertson, G. G. FASTDash: A visual dashboard for fostering awareness in software teams. In *CHI* (Apr. 2007), 1313–1322.
3. Brun, Y., Holmes, R., Ernst, M. D., and Notkin, D. Speculative analysis: Exploring future states of software. In *FoSER* (Nov. 2010), 59–63.
4. Brun, Y., Holmes, R., Ernst, M. D., and Notkin, D. Proactive detection of collaboration conflicts. In *ESEC/FSE* (Sep. 2011), 168–178.
5. Collins-Sussman, B. The Subversion project: Buiding a better CVS. *Linux 2002*, 94 (2002), 3.
6. de Souza, C. R. B., Redmiles, D., and Dourish, P. "Breaking the code", Moving between private and public work in collaborative software development. In *GROUP* (Nov. 2003), 105–114.
7. Dewan, P., and Hegde, R. Semi-synchronous conflict detection and resolution in asynchronous software development. In *ECSCW* (Sep. 2007), 159–178.
8. Estublier, J., and Garcia, S. Process model and awareness in SCM. In *SCM* (Sep. 2005), 59–74.
9. Grinter, R. E. Using a configuration management tool to coordinate software development. In *CoOCS* (Aug. 1995), 168–177.
10. Grune, D. Concurrent Versions System, a method for independent cooperation. Tech. Rep. IR 113, Vrije Universiteit, 1986.
11. Hattori, L., and Lanza, M. Syde: A tool for collaborative software development. In *ICSE Tool Demo* (May 2010), 235–238.
12. Holmes, R., and Walker, R. J. Customized awareness: Recommending relevant external change events. In *ICSE* (2010), 465–474.
13. Kersten, M., and Murphy, G. C. Using task context to improve programmer productivity. In *FSE* (2006), 1–11.
14. Milewski, B. Distributed source control system. In *SCM* (1997), 98–107.
15. Perry, D. E., Siy, H. P., and Votta, L. G. Parallel changes in large-scale software development: an observational case study. *ACM TOSEM 10* (July 2001), 308–337.
16. Rochkind, M. J. Mining metrics to predict component failures. *IEEE TSE 1*, 4 (1975), 364–370.
17. Sarma, A. A survey of collaborative tools in software development. Tech. Rep. UCI-ISR-05-3, University of California, Irvine, Institute for Software Research, 2005.
18. Sarma, A., Bortis, G., and van der Hoek, A. Towards supporting awareness of indirect conflicts across software configuration management workspaces. In *ASE* (Nov. 2007), 94–103.
19. Sarma, A., Noroozi, Z., and van der Hoek, A. Palantír: raising awareness among configuration management workspaces. In *ICSE* (May 2003), 444–454.
20. Sarma, A., Redmiles, D., and van der Hoek, A. Empirical evidence of the benefits of workspace awareness in software configuration management. In *FSE* (Nov. 2008), 113–123.
21. Śliwerski, J., Zimmermann, T., and Zeller, A. HATARI: Raising risk awareness. In *ESEC/FSE* (2005), 107–110.
22. Tichy, W. F., and Tichy, W. F. RCS - a system for version control. *Software: Practice and Experience 15* (1985), 637–654.
23. Walrad, C., and Strom, D. The importance of branching models in SCM. *Computer 35*, 9 (Sep. 2002), 31–38.
24. Wloka, J., Ryder, B., Tip, F., and Ren, X. Safe-commit analysis to facilitate team software development. In *ICSE* (May 2009), 507–517.