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.
We imagine future collaborative development tools that not
with others whose future changes are likely to cause conflicts
(ASTs). Two potentially conflicting changes to the same file
devolve into costly conflicts. The need for early conflict
detection, identifying changes that are guaranteed not to cause tests
to fail; this allows developers to share some of their changes
early. 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, Crystal1, creates
copies of the developers’ code and speculatively merges them
in the background using the version control system. Crystal re-
ports 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 — rep-
resents 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.

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:

1http://crystalvc.googlecode.com
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**


