Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development

Prasun Dewan
University of North Carolina
dewan@unc.edu

Rajesh Hegde
Microsoft Research
Rajesh.Hegde@microsoft.com

Abstract. Previous work has found that (a) when software is developed collaboratively, concurrent accesses to related pieces of code are made, and (b) when these accesses are coordinated asynchronously through a version control system, they result in increased defects because of conflicting concurrent changes. Previous findings also show that distance collaboration aggravates software-development problems and radical co-location reduces them. These results motivate a semi-synchronous distributed computer-supported model that allows programmers creating code asynchronously to synchronously collaborate with each other to detect and resolve potentially conflicting tasks before they have completed the tasks. We describe, illustrate, and evaluate a new model designed to meet these requirements. Our results show that the model can catch conflicts at editing time that would be expensive to manage at later times.

Introduction

Complex software must be developed collaboratively. While recently there has been some interest in synchronous pair programming, traditionally the collaboration is asynchronous, with programmers working independently on the same or different parts of the software. Even in pair programming, different pairs work asynchronously on the same project. In asynchronous software development, there is a need for coordination mechanisms to manage conflicts. Traditionally, such mechanisms are provided by version control systems, which require programmers to individually address the conflicts at check-in time. Inspired by the

findings that distance collaboration aggravates software-development problems (Herbsleb et al. 2000) and radical co-location reduces them (Teasley et al. 2000), we identify a new distributed computer-supported model of software development that provides semi-synchronous conflict-management in asynchronous software development. By conflict management we mean determining if there is a conflict, identifying how to resolve it, and performing the fix. By semi-synchronous collaboration we mean a mix of synchronous and asynchronous collaboration.

The general concept of breaking collaboration into diverging asynchronous and converging synchronous phases has been presented in previous work (Munson et al. 1994). Here we consider a concrete realization of this concept in which the synchronous phases are used only for conflict management. Lightweight system-provided mechanisms are used to make transitions between the two phases.

To investigate this and other ideas, we have extended the user interface of the Visual Studio software development environment – we call the extended user-interface CollabVS. The design, implementation, novelty, and all possible uses of the extensions are not a focus of this paper. In fact, some of the extensions are also provided by recent programming environments and can be easily improved. Here we focus on the narrower issue of the application of these mechanisms in developing a new conflict-management model.

The rest of the paper is organized as follows. We first derive the collaboration model based on the results of previous research. Next we present a simple but realistic joint programming example to illustrate the model. We then identify a joint software development task also designed to exercise the model that is more elaborate than the example but small enough to be carried out in a lab study. Next we present the actual study performed using the task, and end with conclusions and future work.

Deriving the Model

In this paper, we will talk about both previous work and our own contribution at the model level. A model abstracts out details of the user activities supported by a single tool or a set of integrated tools. As these activities are supported directly by the tool (set), we assume there are lightweight mechanisms to transition among them that do not require the use of the OS to explicitly start applications. By abstracting out tool details, it is easier to reason about them and improve their shortcomings. In fact, in this section, we will derive our conflict management model by identifying and refining the models supported by previous work on collaborative software engineering tools. Before we do so, let us first identify the problems these models address.

Brooks (1974) found that adding more people to a software team does not necessarily increase the productivity of the team because of coordination costs. This observation seems unintuitive for two reasons. First, documentation should re-

duce the need for direct communication. Second, modular decomposition of software products should isolate software developers. However, studies have found that documentation and partitioning approaches do not work in practice. Curtis et al. (1988) found documentation is a not a practical alternative because requirements, designs and other collaborative information keep changing, making it hard to keep their documentation consistent. After finishing an activity, software developers often choose to proceed to the next task rather than document the results of what they have done. Perry et al. (2001) studied Lucent's 5ESS system and found a high level of concurrency in the project - for example, they found hundreds of files that were manipulated concurrently by more than twenty programmers in a single day. Often the programmers edited adjacent or same lines in a file.

Version control systems, when used in conjunction with programming environments, address the problem of concurrent accesses. After programmers have completed an editing task to their satisfaction, they switch to the version control system (window/perspective/tab), check in their changes for all programmers working on the project, and use the diff tools of the system to identify conflicts. If no conflicts are found, they can end the task. Otherwise, after viewing/processing one or more potential conflicts reported by the version control system, they can check-out the code, switch to the editing system (window/perspective/tab), and fix the real conflicts to carry out another iteration of this process. As mentioned above, this process involves editing and conflict detection phases, all of which are carried out asynchronously by the programmers, though they may use check-in notifications (Fitzpatrick et al. 2006), email, IM, virtual "ticker tapes" (Fitzpatrick et al. 2006) and other communication mechanisms to trigger synchronous collaboration supported by some external tool that is not integrated with the version control system.

Even though this model provides conflict management, (Perry et al. 2001) have found it does not work well. They found a positive correlation between the amount of concurrent activity and defects in a file, despite the use of state-of-the-art version control mechanisms to find and merge conflicting changes. Based on other studies, it is possible to derive some of the reasons for this situation. Programmers do not accurately document their planned and finished tasks and do not look at such documentation. As one programmer put it, "I will just blast ahead and cross my fingers and hope I have not screwed up" (Grinter 1998). Thus, programmers are not able to prevent conflicts themselves during the editing phase (as opposed to check-in time) of their activity because of *insufficient information about the activities of others.*

Another reason for the current problems is that conflicts are detected by a *file-based* diffing tool. Such a tool can only detect direct conflicts, that is, conflicting changes to the same file. Even then, it can give many false positives and negatives because it does not know the structure of the file. It cannot detect indirect con-

flicts involving different files. Moreover, few people have a sense of the overall picture or the broad architecture (Curtis et al. 1988; Grinter 1998), which is required to prevent conflicts. One way to reduce indirect conflicts is to have well-defined APIs between the various components of the system. However, API's may change (Grinter 1998). In fact, new people may be hired simply for the task of adapting to concurrent changes to a new API (de Souza et al. 2004).

Yet another issue with the traditional model is that the conflicts are detected at *check-in* time after a user has performed the task, rather than earlier, when the task is being performed. Because programmers do not have the benefit of a "stitch in time," the repair is costly, leading to the productivity problems reported by Brooks.

Finally, *w*hen two programmers make conflicting changes, the person who checks-in or saves later is responsible for detecting and repairing the conflict *individually*, though, as mentioned above, he/she can use informal channels to involve others in synchronous or asynchronous conflict management. In fact, programmers concurrently working on different private spaces (created from the same base) often race to finish first to avoid having to deal with merging problems (Grinter 1995) and/or re-run test suites on the merges (de Souza et al. 2004). The fact that distance collaboration aggravates software-development problems (Herbsleb et al. 2000) and radical co-location reduces them (Teasley et al. 2000) implies the need for lightweight channels for allowing programmers to collaborate with each other more closely to prevent and resolve conflicts.

Thus, the studies above motivate a new collaboration model that meets the following requirements:

Early conflict detection: Conflicts should be caught while programmers are implementing their tasks rather than at check-in time after they have finished their tasks.

 Dependency-based conflict notification: The system should use information about the dependency among program elements in checked-out versions to notify programmers about both direct and indirect conflicts.

 Collaborative conflict detection and recovery: Ideally, the system should automatically find all conflicts, but this is impossible, in general, because of the halting problem in computer science, though heuristics could be used to do semantic merging in special cases. Therefore, it should provide mechanisms for programmers to collaboratively detect and fix conflicts.

Usability: A model supporting the above features is bound to be more complex than the existing model. Therefore it is important to additionally require the model to be usable. This implies that the model should be easy to learn and provide few false positives about potential conflicts, and the programmers should find each of the activities of the model useful and should not feel that the synchronous phases of the model violates their privacy.

Based on the previous works cited above, we take these requirements as axioms, though (Gutwin et al. 2004) present a study of three open-source software projects that seems to contradict these other works. Based on interviews with fourteen people working on these projects, they found that the developers were hard pressed to recall examples of duplicated or conflicting work. One can argue that that this study does not necessarily contradict the other findings above as open source projects are different from other projects in that they are more loosely coupled, do not have as firm deadlines, and make all information public. Thus people are more in control and aware of the software development process, and hence can better prevent conflicts. Our work and that of several other projects described below is based on the assumption that conflicts do occur, as reported by other papers addressing this issue.

To the best of our knowledge, ours is the first work to explicitly state and derive the set of above requirements, though some subsets of it have been the implicit design goals of many projects. One approach to support early conflict detection is to synchronously show the activities of co-developers. This can be done by showing the exact concurrent edits of collaborators interacting with a synchronous non-WYSIWIS editor (Dewan and Riedl, 1993; Cook et al 2005), or by continuously displaying diffs between different versions of a file (Minor et al. 1993). A user study found that concurrent synchronous editing done by pairs of programmers can, in fact, reduce conflicts and task completion times (Cook et al 2005). (This productivity gain is consistent with the studies of pair programming using a WYSIWIS editor.) However, the also study found that users wished to have the option of disconnected workspaces to work privately. The results of the study imply that it would be useful if changes made to private workspaces could also get the benefit of early conflict detection. Therefore, other systems such as (Hupfer et al. 2004; Josephine Micallef 1991; Cook et al. 2005; Molli et al. 2001; Schummer et al. 2001) show more abstract information about remote activities such as editing of the same checked-out file or method. However, they do not provide enough code contexts to find faults. Moreover, none of the previous semi-synchronous systems provide special code merging mechanisms to fix the conflict before check-in time. In fact, when a potential conflict was identified, users of Tukan (Schummer et al. 2001) resorted to pair programming – in other words, moved to a completely synchronous collaboration model.

Based on the previous findings and system designs, we have developed and evaluated a new semi-synchronous model, in which the editing phases are always asynchronous and the conflict detection and recovery phases may be synchronous or asynchronous. Of course, as mentioned earlier, an asynchronous phase may be executed synchronously by a team of programmers working independently from other programmers. This model is shown graphically in Figure 1. Not all transitions are shown to reduce clutter. Though not shown in the figure explicitly, as in the traditional model, after finishing their tasks, programmers can transition to the

version control system to check-in their changes and detect and fix additional conflicts.
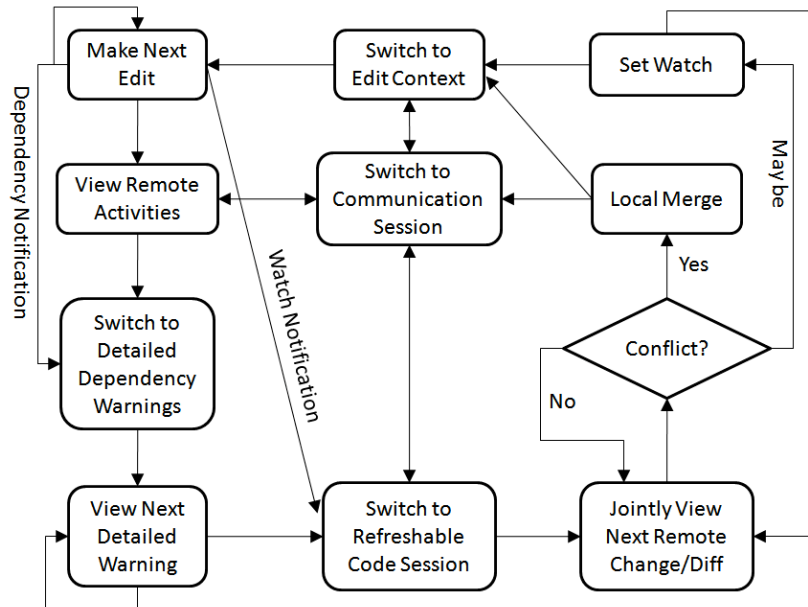

Figure1: Semi-Synchronous Conflict Resolution Model

In our model, as in Jazz, JSE and SubEthaEdit, programmers (a) can be constantly made aware of the program elements accessed by others, which include not only files, as in the previous systems, but also methods and classes, and (b) can easily switch to communication sessions involving team members, which can include not only an IM session, as in the previous systems, but also an audio/video session. Also, as in Mercury (Josephine Micallef 1991) and Tukan (Schummer et al. 2001), they get warning about potential conflicts based on dependency checking among different checked-out versions. The remaining parts of the model described below are entirely new.

Programmers can control the level of synchrony in the conflict detection phase by requesting that dependency checking be delayed by a specified number of edits. They can also choose the granularity of program elements for which dependency checking is done (e.g. method, class, file). Based on the coupling among their tasks, different granularities would be appropriate. It is important to not choose a higher-granularity than appropriate as programmers would then get unnecessary false positives. Moreover, our model requires that the same warning not be displayed twice. Our expectation is that on the first conflict between two program elements, programmers would communicate the work they plan to do on these elements in their checked-out versions, and thus would not need to resolve conflicts between the methods multiple times. Based on the monitoring and warnings, programmers can switch to a conflict inbox, so called, because it can be

considered a persistent collection of detailed conflict messages regarding the current project. As with a regular inbox, the user can iteratively look at each of the items.

After viewing one or more warnings, they can either go back to the edit context, perhaps adapting their work to reduce conflicts, or switch to a code session that can be shared with other programmers. In this session, they can browse the checked-out versions of a remote programmer to identify potential conflicts. Because these versions have not been checked-in, the code in it may not be complete enough to determine if it indeed conflicts with the code of the local user. In this case, the local programmer may set a watch asking the programming environment to inform him/her when the remote user finishes editing a program element, assuming the remote programmer agrees to such monitoring of his/her activities. The local programmer may then switch to editing tasks that are less likely to conflict with the remote developer's current activity. When a watch notification arrives, the programmer can revisit the code session, and continue with the process of identifying conflicts.

Once a real dependency between a local and remote version has been identified, remote changes can be incorporated in the local code to prevent future conflicts. This merge is different from the kind of merge in version control systems in that it affects the local editing buffer rather than a global checked-in version. As these changes are made to the editing buffer by the system to resolve an identified conflict, in our model, they do not trigger conflict warnings.

At any stage in this process, programmers can switch to any of the boxes shown in Figure 1 that are labeled with a title that begins with "switch to". In particular, at any point they can switch to a communication session to identify and resolve conflicts. These boxes represent areas of the screen (such as windows/tabs/panels) that can be displayed/viewed at any time using lightweight commands such as change tab. We have not shown arrows from all tasks to these boxes to reduce clutter.

By simply looking at the model design, it trivial to see that the model meets the requirements of early conflict resolution, dependency-based conflict notification, and collaborative conflict detection and recovery. To determine if it meets the usability requirement requires a programmer study described later.

The model ignores the exact approach for session-creation, notification, dependency checking, diffing, merging, determining when programmers have finished editing a program element, and refreshing code sessions in response to remote changes, which are implementation-dependent. To evaluate and illustrate the model, we have had to resolve these aspects. However, as they don't belong to the model, their nature is not a contribution of this paper.

Illustration and Motivation

To motivate and illustrate the various aspects of the model, we present a realistic two- developer programming exercise that is small enough to be described completely and yet rich enough to benefit from all the of activities of the model.

Consider a drawing tool under development. Assume that the current version of the project contains an abstract class, AShapeWithBounds, that represents a geometric shape with rectangular dimensions. It declares four variables, x, y, width and height, which define the location and size of the shape. It also has the following constructor to initialize the variables:

```
public AShapeWithBounds (int initX, int initY, int initHeight, int initWidth) {
x = initX; y = initY; height = initHeight; width = initWidth;          }
```

Alice is the one who created and checked-in this class. A while after doing so, she realizes that the positions of the height and width parameters should be reversed. She had sorted these two parameters alphabetically – however, the convention is to put the dimension along the X axis before the dimension along the Y axis.[1] Therefore she has checked out the class and is about to correct the ordering. In the meantime, Bob is adding a new subclass of AShapeWithBounds, ARectangle in a separate file. The constructor of the new subclass will call the constructor of the above class to initialize the coordinates and size of the bounding box of the rectangle. As a result the two activities conflict with each other.

We will assume that the developers are unaware of each other's tasks and thus do not know they conflict. This is realistic. Bob may be using the API developed by Alice, and developers and users of the API may not communicate with each other (de Souza et al. 2004). Thus, this is an example of an indirect conflict involving different files rather than a direct conflict involving the same file. As Bob is unaware of Alice's change of mind, his constructor follows the parameter order of the base class constructor in the original version of AShapeWithBounds.

```
public ARectangle(int initX, int initY, int initHeight, int initWidth):base (initX, initY,
initHeight, initWeight){ }
```

This code will not work correctly with Alice's new version of the base class constructor. When the collaborators rely only on the version control system, even one that is fine-grained, to coordinate their changes, the earliest point at which they can detect the conflict is when the later user commits. As the changes occur in two different files, the version control system, in fact, cannot detect the conflict. As the original and changed constructors have the same signature, ARectan-

---

[1] This is a mistake the first author actually made, and like many errors, seems uncharacteristic of a proficient programmer only in retrospect.

gle, will compile correctly with the new version of AShapeWithBounds, and thus a build will also not catch the conflict. If appropriate testing or code reviews is not done, then the conflict would be caught at usage time.

We show below how our collaboration model supported by CollabVS can help prevent this conflict or catch it earlier without relying on expensive testing or code reviews.

Let us assume that Alice has started changing the constructor parameters of AShapeWithBounds() when Bob starts editing the constructor ARectangle(). CollabVS displays to Bob a user-panel for Alice showing the log-in name of the remote user, whether the user is online, the file, class and method on which the user is focusing (currently has cursor in), and status of the current activity – editing, viewing or debugging (Figure 2). This information is synchronously updated. Bob could look at Alice's user panel to realize she is editing a method he intends to call.

As Bob is not expecting a conflict, it is likely that he does not actually look at the user tile. As soon as he starts editing the constructor, however, CollabVS automatically detects a potential conflict. In general, it 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. It looks for dependencies among three kinds of program elements: file, type (class or interface), and method. Each of these program elements depends on itself. In addition, a type depends on a subtype and supertype, and a method depends on a method it calls or is called by. Such dependencies extend recursively beyond one level. For example, a subtype may have another subtype of its own, and CollabVS works at any depth of such dependencies.
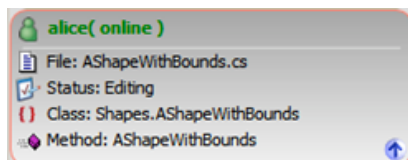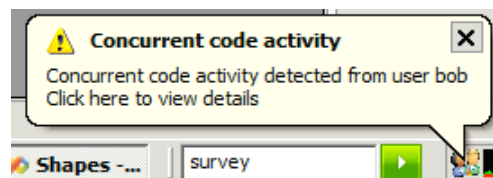

Figure 2: Coding Awareness.

Figure 3: Notification Balloon (Alice's View).

On detecting a conflict, CollabVS displays a notification balloon that gradually fades away (Figure 3) so that, in case of a false positive, programmers can ignore it much in the way they ignore junk-mail notifications today. (In fact setting the dependency-checking parameters can be expected to be similar to the process of defining junk mail filters.) Clicking on the notification balloon automatically takes the user to the conflict inbox (Figure 4) displaying a persistent collection of detailed conflict messages regarding the current project. In general, the person whose edit created the conflict is responsible for initiating the (possibly collaborative) resolution of the conflict. This means that Bob must decide on the next step.
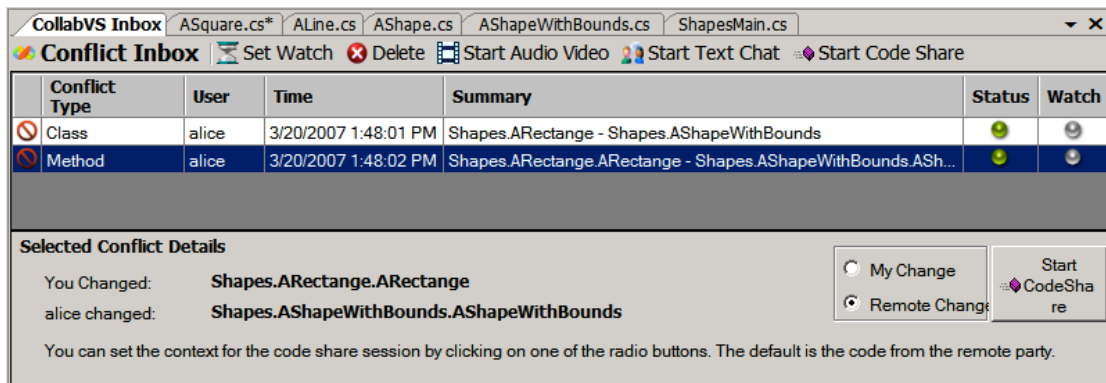
Figure 4: Conflict Inbox

Bob realizes from the notification and inbox that Alice is in the middle of changing the called method. He has other changes to make, and therefore, decides he should delay editing the constructor until Alice has decided what she wants to do with the called method. Therefore, he sets a watch on the conflict by selecting the conflict (Figure 4) and pressing the *set watch* command. Based on the selected conflict, CollabVS knows the dependent program element edited by the collaborator. It gives the active user the option of waiting until the collaborator moves away from editing the current program element and starts editing a different one. As a user may make a temporary movement from a method, it also gives the option of waiting for a certain time period (Figure 5). In this example, Bob knows that the constructor on which Alice is working is simple, and infers that any movement from it will probably be permanent. Therefore, he chooses the first, *by code context*, option (Figure 5), filling in some text explaining to Alice why her actions must be tracked by him. When he commits the dialog box, Alice gets a confirm-notification asking if she is OK with Bob setting a watch on her activity, as she might have privacy concerns. When Alice accepts the watch, Bob gets a success message. Once Alice moves the insertion point out of the constructor AShapeWithBounds, CollabVS notifies Bob.

As mentioned before, ColabVS only knows that a potential conflict exists between the two methods – to determine if it is an actual conflict, Bob must actually see what Alice has done. To do so, he selects the conflict and invokes the *start code session* command in the conflict inbox. This command automatically defines a potentially collaborative session for viewing the contents of Alice's conflicting remote program element – Alice's constructor. A code session (Figure 8) shows the local and remote version of the (manually or automatically selected) program element side by side. If Bob had selected the *my change* option in the conflict inbox, then the code session would have shown his conflicting program element in comparison with Alice's version of it. The code session would also be created in Alice's programming environment if the *collaborative code session* setting was chosen. To locate the exact change, he goes to the code-session panel and exe-

cutes the *show diff* command to show the difference (Figure 6) between two program elements (method, in this case). Bob is now easily able to find the change and adjusts the parameters of his own constructor when he codes it. As mentioned in the model description, when he re-edits the constructor he does not get a new conflict notification.
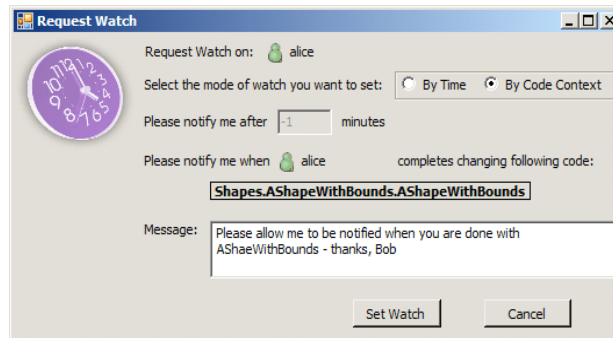

Figure 5: Request for completion Notification

Figure 6 Diff-ing in the Code Session

Once Bob finishes his constructor, he needs to test it. However, the code he has written assumes Alice's version of the AShapeWithBounds() constructor, while his workspace has the old version. Thus, he uses the local merge facility provided by the model to merge his version of the constructor with Alice's version of it to test his code. Currently, CollabVS provides a very simple implementation of this facility that simply replaces the local program element with the remote version of it. Since the code displayed in a code session is a snapshot at the time of starting the session, users can execute the *refresh* command on the code session to get the latest remote version of the program element. Typically such a command would be executed in response to a watch event indicating that the remote user has finished editing the program element that must be imported.

Communication Sessions

We assumed above that Bob had other activities to do when the conflict was detected, and thus could postpone editing of the ARectangle() constructor until Alice had finished editing the AShapeWithBounds() constructor. If this is not the case, he cannot use the code sharing session as Alice has not yet completed her

changes. Therefore, he can use a chat, audio, or video session to communicate with her to determine her intentions. A communication session is shown in-place within the programming environment (Figure 7), so that switching to it is light-weight, as required by the model As Bob does not expect a complicated conversation, he simply opens a text channel. Alice quickly tells him what she intends to do, and Bob adapts his code. When she finishes her edits, he would like to test his code with her version. Therefore, he selects the conflict from the conflict box and sets a watch for when she finishes editing it. When he gets the completion notification, he goes through the process outlined above involving the use of a code sharing session. Thus, users could wait for their collaborators to finish editing a method both (a) to determine if the latter have completed conflicting code, and (b) import remote changes into their workspace.
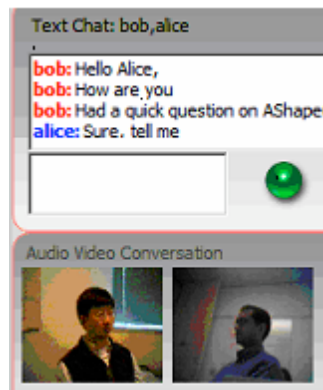

Figure 7: In-Place Communication Sessions

As we see above, the implementation of the model in CollabVS is able to give early and more sophisticated notification about possible conflicts in comparison to a file or version-control system. In a file or version-control system, however, the interleaving of the actions of the two programmers and their tasks does not matter. In our collaboration model, as we see above, it matters, because of the support for early conflict management. When Bob had other activities to perform, he did not need to start a communication session with Alice to determine her intent – he could simply look at her finished method before writing his dependent method. Our model is able to accommodate both schedules.

There are other variations to the scenarios above that are also supported by the model: Bob might edit the ARectangle() constructor before Alice edits the AShapeWithBounds() constructor. In this case, the conflict would be detected when Alice starts her edits, and she would be responsible for managing it. She can start a communication session with Bob and tell him how to adapt his code. Again, he can set a watch on her activity, incorporate her code into his version, and use it to test his code. In all of the cases above, Alice may also incorporate

Bob's changes into her version, or let the version control system correctly merge the two versions.

When a conflict is detected with uncommitted code of another user, the latter may not have an open CollabVS session. In this case, traditional communication channels must be used to contact the other user, which are external to the model. These channels can also be used to completely resolve the conflict – however, for complicated cases, a CollabVS session can be opened by the second user, and then conflict resolution can occur as explained above.

Finally, when a user creates a potential conflict, the other user may have already checked in the code involved in the conflict and exited CollabVS. In this case also the model and its implementation in CollabVS will report the conflict. For each user, the model tracks all concurrent edits since that user checked out a project, including both checked-in and uncommitted edits. These edits persist until the user commits the project to the version control system.

Evaluation

As mentioned above, the assumption behind our work is that the requirements described earlier are axioms as are the findings that motivate them. In particular, we assume that many conflicts occur that cannot be automatically merged in traditional version-control systems. We do not try to provide new research to motivate these requirements, relying entirely on previous work. What we wished to determine is that when conflicts do occur, how usable is our model to detect and resolve them at edit time. This, in turn, requires us to determine if users can easily transition among the various activities of the model, whether these transitions help users identify and resolve conflicts, whether users feel the model violates their privacy, whether the model creates an unacceptable level of false positives, and whether it is easy to learn. To provide preliminary answers to these questions, we decided to conduct a lab study. This task was particularly challenging because it involved multiple distributed programmers working together and we had to assign problems that were "realistic," led to conflicts, and consumed about an hour of work.

We looked hard for examples and/or characterization of such conflicts in the published literature, but were unsuccessful in our search. About this time, the first author was building a whiteboard application in incremental steps. These steps were taken alone and serially by the author, but we tried to determine if the above kind of conflicts would occur if pairs of these steps were taken in parallel by different programmers. We found several pairs of tasks that would cause such conflicts including the motivating example given in the previous session, and chose one of these pairs for our study. One of the tasks in the pair was to add a new feature while the second one involved refactoring existing code for extensibility. The first task is consistent with the practice of continuously adding features (Perry et

al. 2001) while the second is consistent with modern programming philosophies (such as extreme programming) espousing constant refactoring, which is used, for instance, in the development of Mozilla (Reis et al. 2002). We invited pairs of programmers for a lab session and assigned each person one of the tasks in this pair.

Both participants in a pair were given a whiteboard implementation that displayed circles, points and squares. When this program was written, it was assumed that a shape should be oblivious to the display task, which is carried out by a special view class. Unfortunately, this approach has its own disadvantages – the view class used the C# *is* operator to check the class of a shape to determine how to display it. This makes the task of adding new shapes error prone – it is easy to forget to check for the new kind of shape. Therefore, one of the participants ("B") was asked to re-factor the code so that the display operation is implemented by each shape. The other participant ("A") was asked to add a new kind of shape, a line, to the whiteboard. A was responsible for processing input commands to create a line, storing its coordinates, and displaying it. The code session of Figure 8 shows the original code and actual changes made to it by one of the pairs of users. As we see in this picture, a version control system, even a fine-grained one, cannot correctly merge the two sets of changes, which requires that the displaying of a line be done in the model class that stores the line coordinates, and not the view class. Both direct conflicts involving the view class, and indirect conflicts involving the view and model class, occur in this exercise. The goal of our study was to determine if the users were able to easily use our mechanisms to detect and fix the conflict at editing time. We could not evaluate aspects of the model that interface it with traditional conflict management such as email and version control systems as the level of asynchrony in such conflict-management cannot be realistically simulated in a lab study. As mentioned earlier, the model allows programmers to choose the granularity of dependency checking. In the evaluation, we turned on the finest-granularity checking. We did not evaluate the usage of dependency-checking parameters in this study as they depend on the task and programmers' previous experience with the tool, and the participants used out tool for the first time to perform a single task.

We recruited 16 participants from a pool of developers from Microsoft. The pool of participants was gender balanced and included a mix of people with intermediate, advanced, and expert software development skills. Participants were randomly grouped into pairs, assigned to separate rooms, and told that they would be co-workers for the duration of the study. The study involved training participants to ramp them up to speed in getting familiar with the system for about 20 minutes. Then they were given about 60 minutes to complete as much of the assigned tasks as possible. In the end, 10-15 minutes were used to fill in a survey and for debriefing. We recorded participants' actions using LiveMeeting.
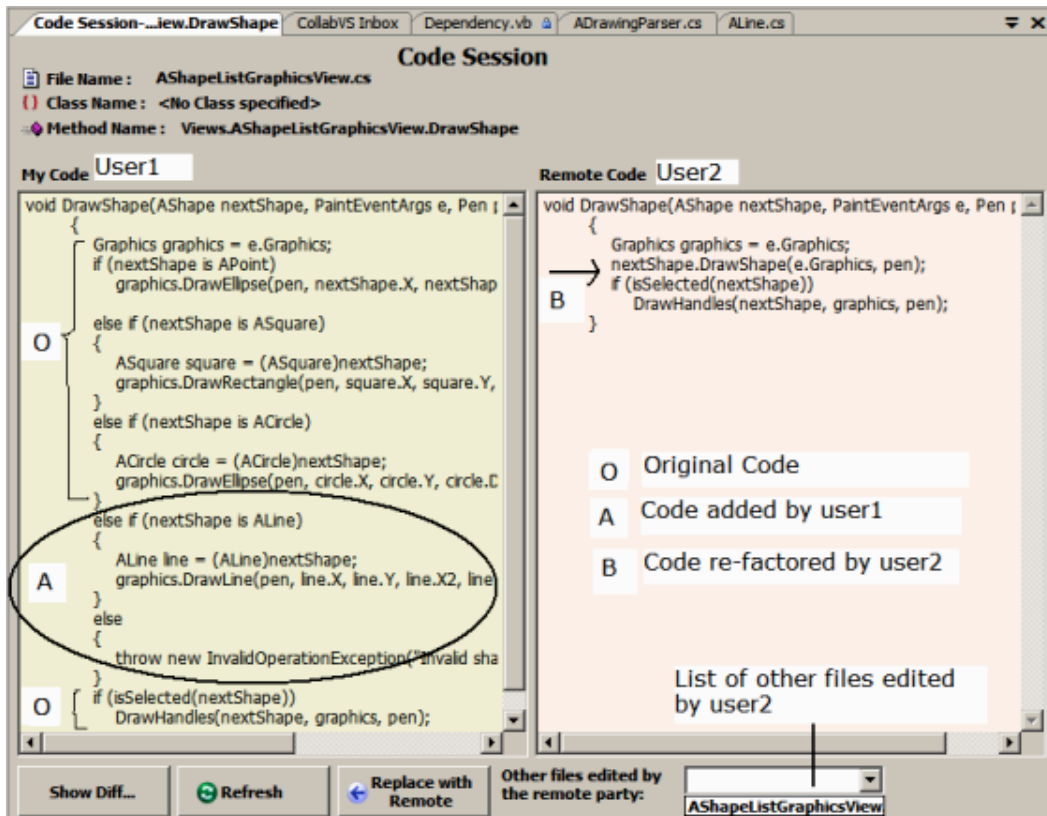
Figure 8: Example of Actual Code Session in Lab Study

Table 1 shows the result of survey questions we asked the participants at the end of their tasks. All survey questions were answered using a 7-point scale where 1 = "strongly disagree" and 7 = "strongly agree". We interleaved positive and negative questions in the survey so the participants did not follow a specific pattern in answering. As mentioned earlier, several programming environments such as Jazz support monitoring of the focus of a collaborator. Telling others the exact method a person is working on goes beyond existing awareness systems. The responses to the first question show that users were not bothered by this invasion of their privacy because, as indicated by responses to the second question, users felt they were able to better plan their work. Not surprisingly, users found it easy to start a communication session with collaborators as lightweight commands were available from the programming environment to transition to them. More interestingly, they found synchronous communication sessions useful to resolve conflicts. They agreed that the collaboration model found conflicts that would have been hard to find otherwise, and interestingly, were mostly not distracted by false positives. In fact, it was possible to do better with false positives. The model allows users to set conflict detection be delayed by a specified number of edits (we had set this to zero for the study). Moreover, the implementation of the model in CollabVS does not currently use program slices to detect conflicts

(Gallagher et al. 1991) as such a capability was not available to us. Finally, Table 1 shows that users found code sessions and watches useful and usable.

	Survey Question	Mean	Med	SDEV
Code-awareness related questions				
1	I was **NOT** comfortable with others seeing my presence information.	2.00	1	1.55
2	Knowing what method or type(class/interface) my co-developer was editing helped me plan my work better	5.50	6	1.51
Communication-tools related questions				
3	It was easy to start and end a conversation(audio, video, text) with my co-worker	6.63	7	0.72
4	It was NOT useful to start an audio, text, video conversation.	1.75	1	1.18
5	I liked having the audio/video/text conversation tools integrated within Visual Studio rather than as a separate tool such as a general IM application.	6.19	7	1.22
Code-conflict-detection and notification related questions				
6	ColabVS helped me find conflicts that would have been hard to find otherwise.	4.84	5	1.69
7	Automatic conflict detection gave too many false positives and thus was **distracting**.	2.31	2	1.08
8	The fact that I was able to see my co-developer's code (using the code share session) was very useful to me to resolve conflicts.	6.16	6.5	1.12
9	It was easy to use code share session	5.97	6	1.16
10	It was **difficult** to understand the conflict notifications that the system provided.	3.13	2.5	1.54
11	It was easy to set a watch on my co-worker's activity	5.69	6.5	1.48
12	It was useful to set watch on my co-worker's activity	5.31	5.5	1.57
13	Setting a watch helped me concentrate more on my work than polling for information through CollabVS awareness information.	5.38	5.5	1.53
General questions				
14	Overall, it was easy for me to use CollabVS to collaborate with my co-worker	5.66	6	1.35
15	Overall, the CollabVS tool window was **distracting**	2.63	2	1.82
16	Overall, I liked working with CollabVS ON in my Visual studio (than working without it)	6.00	6	1.03

Table 1: Survey Questions and results (1: Strongly Disagree, 7: Strongly Agree)

Some participants did not answer questions about the *watch* feature because they did not use it (9 out of a total of 288). We ignored their answers in the calculations. Based on the LiveMeeting video recordings, we were able to determine that text chat, audio conversation, watch, asynchronous code session, and synchronous code session were used by 4, 6, 4, 5, and 6 teams, respectively. This im-

plies that these aspects can be learnt in about twenty minutes of training time – as we see each feature was used by at least half of the participants. Thus, we believe the results show that these features of the model are easy to learn, easy to use, and useful. This data also shows that synchronous code sessions were used by all participants and some also used asynchronous code sessions. The former were used in deciding how to fix the conflict and the latter to merge changes. We must rely only on the subjective answers of Table 1 to determine the usefulness of monitoring others' activities, which cannot be determined by examining the videos.

Overall, users said they were happy with the features of our model and were not distracted by the CollabVS window. This is not only reflected in the answer to questions 14-16, but also the free-form comments they filled at the end of the survey and remarks during debriefing, some of which are reproduced below: "Watch is extremely useful when people have different working hours.'' "This is really cool system for a small team." "Very useful not just for code, but also for SQL queries, documents etc if supported." "It's really nice to know when someone is about to add some constructor, about to make a method virtual, about to add new class etc" "Would love to have it now." "This tool has the ability to spark discussion rapidly and get to a decision really quickly. This has got to be great money and time saver. It not only keeps the project cost down but also leads to higher quality software." "I first thought it would be annoying to have all these awareness. But as I used them, I loved them." Some participants had a few suggestions as well. "While the tool is definitely useful, it has the potential to "bug" others. A do not disturb mode will help reduce this." "I would have liked to see the popups not steal my focus." "I do not like popups, it's very easy to just dismiss it without seeing. Some integration with messenger may help, or some notification in the sidebar would be great." Popups are provided for accepting watch and audio/video session requests. "Conflict detection inbox takes away space from the code editor. Maybe this should be a window at the bottom." "I would have loved to see what code share my co-worker is watching right now." "Good to know which ones (files) I already copied from a remote version and which ones I didn't". Some of these negative comments can perhaps be overcome in a better implementation of our model.

Thus, this initial user study shows that our model is promising and worth more investigation. It is possible that the participants were primed by the tutorial to think about conflicts. However, this does not detract from the results as our goal was to determine how easy it is to catch conflicts with our model at editing time that cannot be caught by the traditional model, and not to determine if conflicts actually occur. The programmers were not told in advance what the conflicts were.

All of the pairs did indeed detect and completely fix the conflicts in the time period. The tasks would have taken less time had the programmers not had to detect and fix the conflicts while programming, but the checked-in code would have

had faults. Even if all of the 60 minutes were used for conflict management, our results are encouraging because this time period is about an order of magnitude less than the average time required to fix a bug - the literature reports it to be between 5 and 15 hrs (Humphrey 1997). We did not directly compare how well the given problem-pair would be solved with and without our tools because (a) The bug detection and fixing time of 5-15 hrs given above is much more than the time available for a lab study, and (b) all participants had used Visual Studio without our tools and thus could give informed comments about the benefits and drawbacks of the tools. Thus, we assumed the earlier findings about the cost of late conflict management, and in our study were interested only in determining if our tools were effective in enabling conflict management at edit time. In general, our model can catch conflicts that are also caught by code reviews and other systems (possibly using semantic information) at file-save, check-in or later times. Even in the case of these conflicts, it can prevent programmers from taking actions that depend on the conflict. Moreover, as mentioned in the introduction, our approach could also allow different pairs of programmers to manage conflicts.

Conclusions and Future Work

This paper makes several contributions. It distills earlier work into a set of requirements that have not been explicitly identified before, and presents a new model that meets these requirements and subsumes and extends previous models. Another important contribution of the work are two concrete realistic examples of conflicting coding, a small one used to illustrate the model and a longer one used in the study, that are small enough to be presented in a paper and rich enough to bring out the limitations of the traditional conflict management model.

These examples could be the bases for concrete illustration of future models/tools and motivate other detailed conflicting tasks that serve as benchmarks for conflict-management tools. Perhaps the most important contribution of the paper is an experiment that allowed the use of several conflict-management features, both existing and new, that have never been evaluated before by a programmer study.

Conflicting code is an old and complex problem that will not be solved by any one result. It requires advances in many areas such as visualization and project management not addressed here. The contributions of this paper take us closer to finding the ideal collaboration model to address this problem. They also suggest several future steps towards this goal. It would be useful to perform a lab study involving teams that are larger than two members. Many aspects of our model assume a small team-size – in particular the approach of showing and logging the conflicts with each team member, which scales better than synchronous (WYSIWIS and non-WYSIWIS) programming but worse than traditional asynchronous development. This does not make such aspects impractical because a

team typically has less than eight members (Booch et al. 2002), though it may occasionally have hundreds of members. It would be useful to create scaleable version of our model that accommodates larger teams. Programmers may have been more concerned about privacy if they were doing real work rather than a lab exercise. To answer these and other questions with the evaluation, the most important next step for us is to make the system more robust and perform a field study with it. We can imagine this happening in two stages. In the first one, we would simply detect conflicts without reporting them, thereby obtaining a set of real-life conflicts. These could form useful benchmarks for research in this area. More important, based on these conflicts, we could refine, configure, and turn on our dependency reporting; and determine qualitative data regarding how well the users found the system usable and useful, and quantitative data regarding changes to defect rate. It would be useful to explore the use of some of these features for purposes other than conflict management in software development. For example, a user seeing a collaborator working on a method may ask him a question about the method. Moreover, we have seen here examples of awareness and notification mechanisms tied to the context of software development. It would be useful to explore analogous mechanisms in other contexts such as paper writing. For example, it may be useful to determine the exact section on which a user is working and to get a notification when a collaborator finishes working on a section of a paper. These results can be then used to improve and refine the collaboration model and resolve issues that have been left as implementation-defined in this paper.

Acknowledgments

References

Booch, G. and A. W. Brown (2002). 'Collaborative Development Environments', www.jorvik.com/alanbrown/files/cde-v4.1.pdf.

Brooks, F. (1974). "*The Mythical Man-Month*." Datamation 20(12): 44-52.

Cook, C., W. Irwin, et al. (2005). A User Evaluation of Synchronous Collaborative Software Engineering Tools Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05) - Volume 00 IEEE Computer Society: 705-710

Curtis, B., H. Krasner, et al. (1988). "*A field study of the software design process for large systems*." Commun. ACM 31(11): 1268-1287.

de Souza, C. R. B., D. Redmilles, et al. (2004). 'Sometimes You Need to See Through Walls - A Field Study of Application Programming Interfaces'. *Proc. Computer Supported Cooperative Work* pp. 63-92.

Dewan, P. and J. Riedl (1993). 'Towards Computer-Supported Concurrent Software Engineering '. IEEE Computer 26(1) pp. 17-27.

Fitzpatrick, G., P. Marshall, et al. (2006). "CVS integration with notification and chat: lightweight software team collaboration" *Proc. Computer Supported Cooperative Work* pp. 49-58.

Gallagher, K. B. and J. R. Lyle (1991). "*Using Program Slicing in Software Maintenance.*" IEEE Transactions on Software Engineering 17(8): 751-761.

Grinter, R. E. (1995). 'Using a Configuration Management Tool to Coordinate Software Development'. *Proc. Organizational Computing Systems* pp. 168-177.

Grinter, R. E. (1998). 'Recomposition: Putting it All Back Together Again'. *Proc. Computer Supported Cooperative Work* pp. 393-402.

Gutwin, A. C., R. Penner, et al. (2004). 'Group awareness in distributed software development'. *Proc. Computer supported cooperative work* pp. 72-81.

Herbsleb, J. D., A. Mockus, et al. (2000). 'Distance, dependencies, and delay in a global collaboration'. *Proc. Computer Supported Cooperative Work* pp. 319-328.

Humphrey, W. (1997). *A Discipline for Software Engineering.*Addison Wesley

Hupfer, S., L.-T. Cheng, S. Reiss, J. Patterson (2004). 'Introducing collaboration into an application development environment'. *Proc. Computer Supported Cooperative Work* pp. 21-24.

Josephine Micallef, G. E. K. (1991). "*Extending the Mercury System to Support Teams of Ada Programmers.*" *1st International Symposium on Environments and Tools for Ada* pp. 49-60.

Minor, S. and B. Magnusson (1993). 'A Model for Semi-(A)Synchronous Collaborative Editing'. *Proceedings of European Conference on Computer Supported Coopearative Work.* 219-231

Molli, P., H. Skaf-Molli, et al. (2001). 'State Treemap: An Awareness Widget for Multi-Synchronous Groupware'. Proc. *International Workshop on Groupware.*

Munson, J. and P. Dewan (1994). 'A Flexible Object Merging Framework'. *Proc. Computer Supported Cooperative Work.* 231-242

Perry, D. E., H. P. Siy, L. G. Votta (2001). "*Parallel Changes in Large-Scale Software Development: An Observational Case Study.*" ACM TOSEM 10(3): 308-337.

Reis, C. R. and R. P. d. M. Fortes (2002). 'An Overview of the Software Engineering Process and Tools in the Mozilla Project'. *Proc. Open Source Software Development Workshop.*

Schummer, T. and J. M. Haake (2001). 'Supporting distributed software development by modes of collaboration' *Proc. European Computer Supported Cooperative Work* pp. 79-98.

Teasley, S., L. Covi, M. S. Krishnan, J. S. Olson. (2000). 'How does radical collocation help a team succeed?' *Proc. Computer Supported Cooperative Work* pp. 339-346.