

Dimensions of Tools for Detecting Software Conflicts

Prasun Dewan
University of North Carolina
Department of Computer Science
Chapel Hill, NC 27516, U.S.A.
(01) 919 962 1823
dewan@unc.edu

ABSTRACT

Previous work has found that the number of defects in a source file is proportional to the number of developers who concurrently access the file. Several “conflict-recommender” tools have been proposed that can aid programmers in detecting conflicts that lead to such defects. These can be classified according to several design dimensions including how early in the programming process the (potential) conflict is identified; which, if any, of existing software systems must be extended to create the tool; the granularity of the program constructs that are identified as conflicting; the criteria used for identifying conflicts; how the conflict information is obtained; and whether the tool supports individual or collaborative inspection of the conflict. The various points defined by this design space can be analyzed according to several evaluation dimensions including the number of false positives and negatives given by the tool; how much effort is required to find/fix the conflict; the computation and communication costs of the tool; how much change it requires to the current software development process; how much screen real-estate is used by the tool during coding; and to what extent is the privacy of programmers invaded. The identification and analysis of these design and evaluation dimensions can lead to better evaluation of the various aspects of existing tools and an integrated tool that combines orthogonal features of different tools.

Categories and Subject Descriptors

D.2.6 [Software Engineering]: Programming Environments – *Programmer workbench*. D.2.7 [Software Engineering]: Distribution, Maintenance, Enhancement - *Version control*. D.2.9 [Software Engineering]: Management - *Version control*.

General Terms

Management, Design, Reliability, Human Factors

Keywords

Collaborative software development, configuration management, programming environments, awareness, visualization, notification

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

RSSE '08, November 10, Atlanta, Georgia, USA
Copyright 2008 ACM 978-1-60558-228-3 ...\$5.00

1. INTRODUCTION

Complex software must be developed collaboratively. However, Brooks [1] observed that adding more people to a software team seemed to result in a disproportionate increase in coordination cost. This observation seems unintuitive for two reasons. First, documentation should reduce the need for direct communication. Second, modular decomposition of software products should isolate software developers. However, studies have found that the approaches of documenting and partitioning are far from a panacea. Curtis et al [2] found documentation is problematic 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 [3] found that partitioning does not isolate programmers. They studied Lucent’s SESS 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, check in their changes, 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, 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, email, IM, 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 [3] 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. One reason for this situation seems to be that 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” [4]. 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.

Ideally, what is needed is a tool that precisely identifies all conflicts that result from such activities. However, developing such a tool (for Turing-complete programming languages) would be equivalent to solving the halting problem [5]. Therefore, the best we can hope for is one or more “recommender tools” that provide more information than what is provided by traditional version control systems to detect and resolve conflicts. A variety of research tools that meet this requirement have been developed. However, these tools have not been compared with each other using some common evaluation criteria. Moreover, each of them seems to have unique advantages. There has been no effort that explicitly tries to integrate these advantages.

This paper is a first-cut effort at addressing these two related problems. It classifies existing conflict-management tools according to several design dimensions such as how early in the programming process the (potential) conflict is identified. By decomposing complete designs into multiple components, it makes it possible to combine features found in different existing designs into a single new integrated design. It also evaluates the various points defined by this design space according to several evaluation dimensions such as to what extent is the privacy of programmers invaded. The evaluation makes it possible to reason about the usefulness of the various design decisions and also identifies some inherent tradeoffs that prevent all advantages of current designs to be combined.

The rest of the paper is organized as follows. The next section addresses the evaluation criteria. The following section is the bulk of the paper. It identifies the evaluation and design dimensions, uses the design dimensions to classify existing tools, and analyzes these tools according to the evaluation dimensions. The final section identifies opportunities for integrating the designs, and other directions for future work.

2. EVALUATION DIMENSIONS

False Positives and Negatives: Because of the halting problem, a tool that tries to identify conflicts can give false positives and/or negatives. Intuitively, it is not possible to achieve both low false positives and negatives – a tool that is more aggressive about identifying conflicts should lead to fewer false negatives and higher false positives. In the extreme case, a tool that identifies each concurrent edit as a potential conflict will give no false negatives but the maximum possible number of false positives. As it turns out, it is possible to have both high false positives and high false negatives in the case of conflict detection. This is particularly true when we consider current version control systems because these tools rely on line-based diffs to find conflicts. This approach does not detect conflicts involving different files or even indirect conflicts within the same file such as those between a called and calling procedure. On the other hand, it flags additions and deletions of lines in a file as conflicts even if they are semantically unrelated to the changes made by the other programmer.

Effort required to find potential conflicts: A related issue is the effort required to identify conflicts. It is fairly easy to find conflicts using version control systems as the changed, inserted, and deleted lines are precisely indicated. This is not the case in Augur [6], which provides a visualization that shows, for each line of a program, its modification time, author, and length. While the tool is not advertised as a conflict recommender, it does

provide information that can indirectly help determine conflicts. For example, recent changes to method signatures in related files may indicate a potential conflict among them. However, it is up to the visualization viewer to make this deduction. Thus, in comparison to version control systems, it requires much more effort to determine the conflict.

Effort required to fix a conflict: Once a conflict is found, some effort is required to fix the conflict. A tool that flags each concurrent edit as a conflict requires little cost to fix the conflict as the programmers involved in making the conflicting changes can provide a “stitch in time”. In contrast, version control systems require expensive rollbacks of checked-in changes.

Change to the traditional software process: Tools differ also in the degree to which they change the current software development process. Current version control systems, being the state of the art, do not change this process at all. A tool that flags each concurrent edit as a conflict requires the programmers to do something new - check for conflicts - on each remote edit, and thus makes the maximum change to the process. A system like Augur provides an intermediate amount of change as the programmer must now periodically check the visualization created by it.

Privacy invasion: A related issue is that programmers today expect others to see their changes when they commit them as a new version. Moreover, they expect others to see no more than their check-in comments and a diff between the version and others. Tools that make it easy/possible for others to see more information about their changes such as their incremental edits or which lines were last modified by them invade their privacy.

Computation and communication cost: Computation cost includes the cost of comparing conflicting code elements and visualizing the conflicts. Communication costs include the cost of comparing changes made to distributed code elements. Like other evaluation dimensions, these two dimensions also show significant variance. Version control systems have the lowest possible computation and communication costs. Once a version is checked-in to a (potentially) distributed repository – a step required by all tools discussed here – no additional communication is required to compare two checked-in versions. Moreover, computing and visualizing line diffs has insignificant computation overhead. An Augur-like visualization increases the computation load, and a system showing each remote change increases the communication cost.

Conflict-specific screen real-estate during software development: Screen real-estate is an important issue in software development. It is not uncommon for programmers to be surrounded by multiple displays showing various aspects of the development process such as the debugging and program state. Thus, the space used to show conflict-specific information during the software development process is an important aspect of the evaluation. A traditional version control system takes no additional space while the approach of communicating each remote edit requires the code all of programmers concurrently editing a program to be shown simultaneously.

3. DESIGN DIMENSIONS

In the section above, we took three specific designs – current version control systems, Augur, and a tool that shows all concurrent remote edits – to understand the evaluation dimensions and the variance along them. Here, we dissect a greater variety of designs into design dimensions and evaluate each design choice using the evaluation dimensions. The designs we consider include Palantir [7, 8], TreeMap [9][10], active diffs [11], read and edit wear [12], Tukan [13, 14], [15], Jazz [16], and CollabVS [17]. As in the case of Augur[6], some of these tools are general purpose tools to aid collaborative software development and have not been explicitly designed to identify conflicts. We include them because, like Augur, they provide information that may be useful for finding conflicts.

Conflict identification stage: Current tools identify conflicts at different stages in the programming process: edit time [11] [13,14][10][8][16][17], file-save time [9][15], build-time [17], and version check-in time. All other things being equal, the earlier the time at which a conflict is detected:

- The less the effort required to remove the conflict. If the conflict is determined at (a) edit time, then no re-coding is necessary, (b) file-save time, then only conflicting changes made since the last file save need to be undone, (c) build-time, then all conflicting changes since the last build must be undone and (d) check-in time, then conflicting changes since the check-out must be done.
- The more the frequency and amount of network communication. Conflict detection at (a) edit-time requires communication of conflict information on each edit (to the site of all collaborators), even if the (intended or actual) edit is later undone before the next file-save or check-in, (b) file-save time, requires communication of potential conflicts that exist at that time, and (c) check-in time requires no extra communication, as mentioned earlier.
- The more the number of false positives as conflicts that are later undone will be identified.
- The more the amount of screen real-estate required to show the conflict at coding time (as long as the conflict information persists until check-in time). The reason again is that the number of identified conflicts is proportional to how early it is detected.
- The more the intrusion on programmers' privacy as information about edits that are later undone on further thought are communicated to others.
- The more the disruption to the traditional development model in that the increase in the frequency of conflict identification is proportional to how much earlier than check-time the conflict is reported.

Tools extended/created: Some conflict detection tools such as Augur are standalone new ones, while others are extensions of programming environments/editors [8, 11, 13, 14, 16, 17], and file-systems [17]. In general, if a conflict is to be detected at a certain stage of the software development process, a tool supporting that stage or earlier must be extended (by modifying the tool or intercepting its events). In particular, a programming

environment such as Eclipse that provides an interface to file and version control systems can be used to find conflicts at any stage of the development process.

Granularity of conflicting constructs: Conflicts have been identified and detected at various program granularities. Traditional version control systems and active diff [11] detect and report them at the granularity of text characters, as they show the exact differences between two versions (even though they present the differences in units of lines.) Write wear [12] and Augur [6] provide conflict information at the line-granularity, as they do not consider the exact contents of lines. Conflicts have also been identified and reported at the method [8, 13, 14, 17], method-category [13, 14], class [8, 13, 14, 17], file [7-9, 13-15], and directory [9] granularities. All other aspects being equal, the smaller the granularity:

- The smaller the number of false positives and the larger the number of false negatives, as a change at a smaller granularity implies a change at the larger granularities, but not vice versa. For example if the criterion used for conflicts is concurrent editing of the same program construct, then editing different methods in the same class will not be considered a conflict if the granularity is a method, but will be considered a conflict if the granularity is a class.
- The less the effort required to detect the conflict as more information is provided to locate the conflicting program fragments.
- The more the invasion of programmers' privacy because of the additional information provided.

Conflict criteria: A variety of criteria exist for identifying a potential conflict.

- **Concurrent edit status:** Concurrent editing of the same program component is the most popular criterion, supported by almost all conflict-recommender systems, though these systems differ in the granularity supported.
- **Conflict-count:** A generalization of the above approach is to count the number of concurrent edits to the component and allow the programmers to use some threshold number to determine if the accesses indicate a conflict. Molli et al [10] use this approach for the file granularity.
- **Component-count:** A variation of the above approach is to count the number/fraction of subcomponents of a component that have changed. For example, Palantir [7] provides functions that count the fraction of lines and number of interfaces that have changed in a program.
- **Dependency-based:** The above two approaches only detect direct conflicts, that is, conflicts involving the same program component. It is also possible to find indirect conflicts, that is, conflicts involving different components related by program dependencies such as the IS-A dependency among classes/interfaces and the calls dependency among methods [13,14][17][8].
- **Author-display:** A generalization of Augur's line-based approach is to show the last author of a program

construct. If previous authors or others feel the current author may not understand the full impact of the changes to the construct, they can examine the code for conflict.

- **Edit-wear:** A variation of the above approach is to show the number of times a construct has been edited by anyone. Edit wear [12] supports this approach at the line-granularity. Large edit wear may indicate conflict hotspots that should be investigated.

The conflict-criteria dimension, unlike some of the previous ones, cannot be evaluated based purely on qualitative arguments as, because of the halting problem, it is hard to analytically compare the relative effectiveness of all of these techniques. In some of the previous dimensions, it was possible to do so as the differences were a matter of different degrees of some aspect of conflict detection such as the how early the conflict detection was or what the conflict granularity was. Programmer studies are needed to compare all of these conflict criteria with each other.

Communicating conflict information: Information identifying a potential conflict must be communicated to appropriate programmers. As is the case with any kind of information, this information can be pushed or pulled.

- **Push:** Pushing conflict information essentially involves generating a notification for the last editors of the conflicting program components, as in [17]. This approach may interrupt the programmers at an inopportune moment, which is particularly a problem if the programmers, based on context, know that the notification is a false positive. Therefore, it should be easy to ignore such a notification. For this reason, in [17], a fading notification balloon is displayed, which can be ignored like a junk-mail notification.
- **Pull:** Pulling conflict information involves querying for the information at the programmers' convenience. This approach is supported in Augur since programmers view the visualization on demand. While this approach has the advantage of fewer unnecessary interruptions, it has the disadvantages of late conflict identification because communication of the identified conflict is delayed.
- **Awareness:** A compromise is to use an awareness approach, where some area on the screen is continuously updated with conflict information. For example, information about who is editing a file can be continuously updated [16][17]. For programmers continuously monitoring the screen area, this approach reduces to the push approach while for the others it is equivalent to the pull approach. The disadvantage of this approach is that it requires additional screen estate for displaying conflicts during the application development phase. Two schemes can be used to show conflict awareness on the screen:
 - **User-centric:** A special user pane is created for each collaborator. Each user pane contains information about conflicts in which the user is involved. An advantage of this approach is

that the panes can be hidden when users wish to focus on their individual work [17].

- **Object-centric:** The information about conflict involving certain program component is (permanently) attached to the displays of the program components [8]. Whenever the program components are viewed, the associated conflict information is automatically displayed as a side effect. The advantage of this approach is that the information is found at a familiar place – the display of the program component – hence the effort required to locate this information is low. A disadvantage is that in current systems, the conflict information cannot be removed from the display of the associated program component, a problem that should be easy to fix.

Individual vs. collaborative conflict inspection: Once a potential conflict has been identified between two pieces of code, the authors of the code must determine if it is a true conflict. Most designs only provide individual inspection of the conflicting code. Two exceptions are [17] and [18], which allow the authors to collaboratively browse the conflicting code and conflicting information about it. Collaborative inspection can reduce the effort required to process the conflict but has the disadvantage that it more drastically changes the current software process, in which, coding is typically done individually.

4. DISCUSSION

To the best of our knowledge, ours is the first work that identifies the evaluation and design dimensions presented above, and qualitatively compares points along various design dimensions using the evaluation criteria. This is an important contribution for several reasons.

Understanding existing tools: The evaluation dimensions provide a basis for systematically comparing various conflict-recommender designs. The design dimensions provides a way to economically describe and fundamentally understand the large number of existing designs, by focusing on the similarities and differences among these designs rather than describing each design from scratch. The qualitative comparison of the points along various design dimensions, in turn, provides an economical analytical approach to evaluating the whole design space.

Implementing new points in design space: This work also identifies ways to create new designs from existing ones. Specifically, the space identified by the design dimensions has several holes that can be filled by future designs. Page constraints did not allow us to present a large table showing which points are covered by existing designs and which are not. Instead we simply enumerate here some of the unfilled ones. To the best of our knowledge, no current design offers:

- Dependency-based conflict-detection (a) at check-in time, and (b) for granularities smaller than methods and larger than classes.
- An extension of a programming environment that supports conflict-identification at file-save time, which would be more practical than a file-system extension

[15] providing such support as it is easier to change a programming environment (through, for example, plug-ins) than a file system.

- Edit-wear and author-display for components larger than lines such as methods, classes, and files.
- Conflict-count for components smaller than files such as classes and methods.

Rather than only filling holes in the space, it would also be useful to build a single tool that covers the whole design space. For example, it would be useful to build a tool that performs dependency-based conflict detection at edit, file-save and check-in time.

Evaluating tool effectiveness: The evaluation performed here is only qualitative. Previous work has partly addressed this limitation. Two lab studies have found that early dependency-based checking helped programmers resolve and fix conflicts that would not have been detected by traditional tools[17][8]. It would be useful to determine if this holds true in the field and for other points in the design space. This is perhaps the most important future direction for this work. It will help determine which points in the design space are promising and thus should be integrated by future tools.

Evaluating evaluation dimensions: The various evaluation metrics themselves need to be evaluated to determine if they are important. A lab study has found that programmers were not much concerned with privacy issues when others had edit-time awareness of the methods they were modifying [17]. Again, it would be useful to determine if this holds true in field work and for other points in the design space. If it does, then it should be removed from the evaluation dimensions. A similar process is needed for all of the other evaluation dimensions.

Unified framework for software-engineering recommenders: Some of the evaluation and design dimensions seem to apply to recommenders for navigation, refactoring, debugging and other software engineering activities. We can generalize them to dimensions that are independent of the exact programming activity regarding which the recommendation is given such as how much effort is required to determine and process a recommendation, how much additional screen estate does the tool require, what is the computing and communication cost of the recommender, how early is the recommendation made, to what extent are current software practices changed, what is the granularity of the program component to which the recommendation applies, and is pull, push or awareness approach used to communicate the recommendation. It would be useful to explore in-depth a unified framework for designing and evaluating recommender systems.

This paper provides a basis for exploring these future directions.

5. ACKNOWLEDGMENTS

This research was funded in part by NSF grants ANI 0229998, IIS 0312328, IIS 0712794, and IIS-0810861. The comments of the reviewers improved the presentation of the paper.

6. REFERENCES

[1] Brooks, F., *The Mythical Man-Month*. Datamation, 1974. **20**(12): p. 44-52.

- [2] Curtis, B., H. Krasner, and N. Iscoe, *A field study of the software design process for large systems*. Commun. ACM, 1988. **31**(11): p. 1268-1287.
- [3] Perry, D.E., H.P. Siy, and L.G. Votta, *Parallel Changes in Large-Scale Software Development: An Observational Case Study*. ACM TOSEM, 2001. **10**(3): p. 308-337.
- [4] Grinter, R.E. *Recomposition: Putting it All Back Together Again*. in *CSCW*. 1998.
- [5] Horowitz, S., J. Prins, and T. Reps, *Integrating Non-Interfering Versions of Programs*. ACM Transactions on Programming Languages and Systems, July 1989. **11**(3): p. 345-387.
- [6] Froehlich, J. and P. Dourish. *Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams*. 2004.
- [7] Sarma, A., Z. Noroozi, and A.v.d. Hoek. *Palantir: raising awareness among configuration management workspaces*. in *Proceedings of the 25th International Conference on Software Engineering*. 2003.
- [8] Sarma, A., B. G, and A.v.d. Hoek. *Towards Supporting Awareness of Indirect Conflicts across Software Configuration Management Workspaces*. in *Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2007.
- [9] Molli, P., H. Skaf-Molli, and C. Bouthier. *State Treemap: An Awareness Widget for Multi-Synchronous Groupware*. in *CRIWG 2001*.
- [10] Molli, P., H. Skaf-Molli, and G. Oster. *Divergence Awareness for Virtual Team through the Web*. in *IDPT'02*. 2002.
- [11] Minor, S. and B. Magnusson. *A Model for Semi-(A)Synchronous Collaborative Editing*. in *Proceedings of European Conference on Computer Supported Cooperative Work*. 1993.
- [12] Hill, W.C. and J.D. Hollan. *Edit wear and read wear*. in *Proceedings of the SIGCHI conference on Human factors in computing systems*. 1992.
- [13] Schummer, T. and J.M. Haake. *Supporting distributed software development by modes of collaboration*. in *Proceedings of the Seventh European Conference on Computer Supported Cooperative Work*. 2001.
- [14] Schummer, T. *Lost and found in software space*. in *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*. 2001. Hawaii.
- [15] O'Reilly, C., P. Morrow, and D. Bustard. *Improving conflict detection in optimistic concurrency control models*. in *11th International Workshop on Software Configuration Management*. 2003.
- [16] Cheng, L.-T., et al. *Jazzing up Eclipse with collaborative tools*. in *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*.
- [17] Dewan, P. and R. Hegde. *Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development*. in *Proceedings of the 2007 Tenth European Conference on Computer-Supported Cooperative Work - ECSCW*. 2007. Springer.
- [18] Munson, J. and P. Dewan. *A Flexible Object Merging Framework*. in *Proceedings of the ACM Conference on Computer Supported Cooperative Work*. October 1994.