A case for refactoring aware versioning

Torbjörn Ekman
Sven Gestegård Robertz
Dept. of Computer Science, Lund University
Sweden
email: {torbjorn|sven}@cs.lth.se

May 22, 2002

Abstract

The emerging agile methodologies change the way code is developed and, consequently, the type of concurrent changes that are made and which merge conflicts that arise from them. This paper illustrates how version control tools needs to be changed in order to accommodate that new way of developing software.

To give better support for merging code in a process where refactorings are common, we propose an extended representation of differences between versions by adding higher order changes and static semantic information.

We show that this extended version management leads to less false conflicts, more informative conflict reporting, automatic merge of a larger set of changes and more conflicts detected by the merge tool instead of causing compile- or run-time errors.

1 Introduction

The emerging agile methodologies, such as extreme programming (XP)[Bec99, JAH01], promotes practices like collective code ownership, frequent restructuring of the code (refactoring [Opd92]), and continuous integration. This changes the way code is developed and, consequently, the type of concurrent changes that are made and which merge conflicts that arise from them. This paper illustrates how version control tools needs to be changed in order to accommodate that new way of developing software.

1.1 Versioning

In all software projects, it is essential to keep track of the evolution of code and to handle multiple versions of the various parts of the system. It should always be possible to retrieve any previous version of a certain component and to see which changes that lead from one revision to another. This is the objective of version control.

Version control is also necessary to handle the shared data and simultaneous update problems[Leo00] that arise when doing concurrent development. Therefore, tool support for version control is paramount.

Most tools express the difference between two revisions as a set of add, delete, and change operations. An example where one line is changed and one line is deleted is shown in Figure 1.

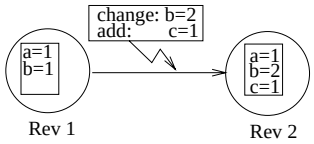

Figure 1: Basic versioning. The differences between two revisions are shown.

In order to support concurrent development, the versioning tool must support *merging* of changes, i.e., integrating two sets of changes to the same file to produce a result containing all the changes. Figure 2 shows schematically how users A and B make simultaneous changes to revision 1 of a file and how their changes are merged to form revision 2.

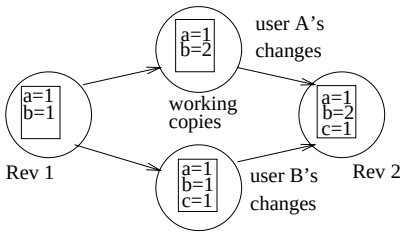

Figure 2: Merging of simultaneous changes by users A and B.

If changes made by users A and B affect the same place in the code, an automatic merge cannot be made. This is called a merge conflict and has to be solved manually. Figure 3 shows an example where a change to and a delete of the same statement are in conflict.

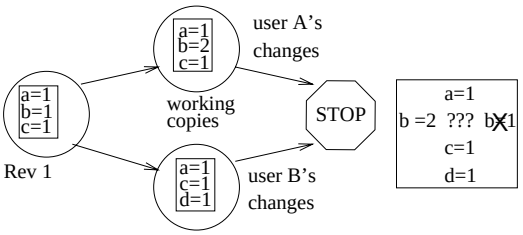

Figure 3: Conflict between the deletion of and the change to the statement `b=1`

1.2 XP practices affecting versioning

In contrast to traditional software development processes, where each developer typically has a well defined area of responsibility, collective ownership encourages developers to make changes anywhere in the code. This increases the probability of programmers making simultaneous changes to the same parts of the code.

Since XP teams do not do a big design up front, the architecture needs to evolve to accommodate requirement changes. Therefore, refactoring is a central part of such software development. A refactoring[Opd92, Fow99] is a controlled structural change – a change that changes the structure of the code but not the behavior.

Integration is difficult, and the longer the time between integration, the more complex it gets. Therefore, extreme programmers integrate their changes as often as possible.

1.3 Limitations of traditional version control

Currently, most commercially available versioning systems are text and line based and work on a per file basis. This works well in traditional projects where strict code ownership and well defined programming tasks keep programmers from doing simultaneous changes to the same files. In such cases, merge conflicts are not very common.

The XP practices change the way code is developed. For instance, collective code ownership makes simultaneous changes to the very same parts of the system much more likely, frequent refactorings cause pieces of code to move around and entities to be renamed a lot, and continuous integration causes merging to be done much more often. Since more programmers work on the same code, merge conflicts are more common. At the same time merging is done more frequently and therefore has to be as simple as possible from the programmer's point of view.

Unfortunately, traditional versioning systems give a lot of false conflicts if extensive simultaneous changes have been made. For instance, if two programmers add a new method at the end of a class, most merge tools will give a conflict since they have added code at the same line in the file. Even worse, if code has been moved in a file, or even between files, a line-based merge tool has no way of producing a satisfactory result since it doesn't know that the set of deleted and added lines it sees is actually a the result of a move operation.

Renaming an entity involves both changing the declaration and all uses of that entity. If a new use of an entity is introduced in parallel to renaming that entity, a text based tool cannot detect that this is a conflict that needs to be resolved. Instead the change will result in a compile-time error.

Furthermore, a single refactoring involves a number of changes but is viewed by the programmer as one logical operation. Hence, in order to increase awareness of changes to the code and helping the user make the right decisions when solving merge conflicts, it should be presented to the user as one, atomic, change.

2 Extended representation of changes

As explained above, traditional versioning systems do not give sufficient support for tightly coupled concurrent development with frequent refactorings. Moved code and renamed entities are not represented properly and this causes false merge conflicts and unclear conflict reporting.

Also, since a refactoring is logically an atomic operation, it would be beneficial to detect that a change is in conflict with a refactoring as a whole instead

of just indicating an ordinary conflict between that change and some part of the refactoring.

To give better support for merging code in a process where refactorings are common, we propose an extended representation of differences between versions.

Higher order changes A set of changes that constitute one, logical operation, e.g., moving a piece of code or performing a particular refactoring can be grouped together. I.e., semantics is added to change sets. This enables the merge tool to present changes at a higher level. For instance,if a `move` and a `change` have been performed in parallel, the differences between the moved and changed lines is displayed instead of producing a conflict between the changed and the deleted lines. This also allows refactoring aware merging, where conflicts based on the semantics of the different refactorings can be detected.

Static semantic information Many refactorings involves changing names, or signatures, of entities. If the tool knows about the semantic connections between, e.g., a declaration and a use of a variable or method, it can detect new types of conflicts; for instance that one programmer added a new use of a certain variable while another programmer removed that variable. It also allows automatic merge in cases where, e.g., uses of an entity that has been renamed are added.

3 Typical refactorings

Many of the refactorings presented by Fowler[Fow99] share the common properties and problems described above. This section presents examples of typical refactorings that illustrate how the extended representation of changes will improve version control.

For each refactoring, a small example illustrates how a traditional merge tool reacts when an ordinary change is done simultaneously with that refactoring. We discuss how this could be improved with our extended representation of changes.

In the examples, we have used the *filemerge* tool from the *Sun Workshop Teamware* suite, which uses a three-way merge. In the figures, an *add* operation is represented by a '+', a change by a '|' and a delete by a '-' sign. The *child* view (on the left) is the locally modified version and the *parent* view (on the right) is the simultaneously refactored version.

3.1 Extract Method

This refactoring is used to extract a piece of code and give it a descriptive name, improving readability and promoting code re-use.

Merge problems occur when the extracted piece of code is changed in parallel with the refactoring. Figure 4 shows how a traditional merge tool behaves in this case. The refactoring is represented as a change to the first line of the extracted code (it is replaced by a method call), deletion of the remaining lines, and addition of the extracted method.

Thus a conflict is detected between the deleted lines in the refactored version and the changed line in the other. It would be more informative to detect a

conflict between the move and change operations. This is made possible by the higher order change *move*.

It might seem trivial to perform an automatic merge (by applying the change deltas to the extracted lines) but, since the extracted lines of code have moved to a different scope this is not always safe.

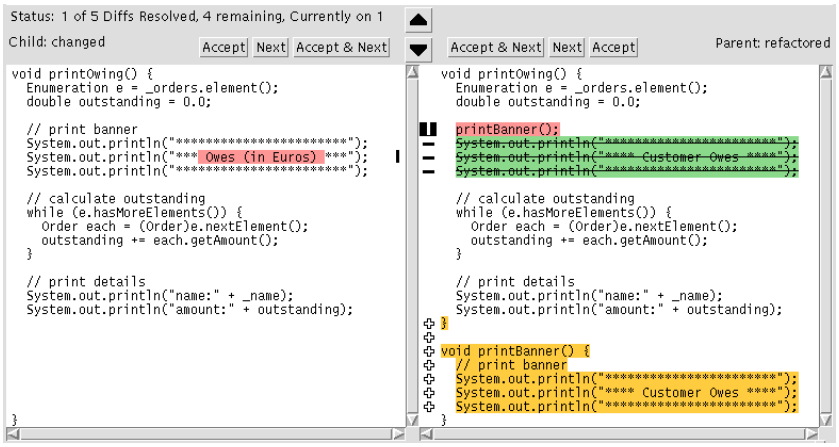

Figure 4: Extract method performed in parallel with a change to one of the extracted lines. A traditional merge tool gives a false conflict between a change and a delete on the same line.

Another case, potentially more dangerous, is illustrated by Figure 5. Here, one of the lines that were concurrently extracted is deleted. This should result in a conflict, but the traditional merge tool only sees that that line is deleted in both versions. This does not cause a conflict and an automatic merge is performed. In the merged version, however, the extracted method still contains the line that should have been deleted and the program doesn't have the desired behavior.

3.2 Move Method

If a method fits better in another class, e.g., because it uses more features of the other class than its current class, the method is moved.

Merge problems occur when the moved method is changed or new invocations of the method are introduced in parallel with the refactoring. Figure 6 shows how a traditional merge tool behaves in that case. The method is deleted in the source class and added in the target class. Then each invocation is changed to reference the method in the target object. A change made to the moved method will results in a conflict with the deletion of the method.

Since the method has moved to another class, the programmer doing the merge only sees that the method has been removed and will probably either put (his new version of) the method back in the old class (leading to two inconsistent implementations of the same functionality in different classes) or discard his changes. Just as in the Extract Method case this can be resolved by knowing that code has been moved.

An added invocation, on the other hand, will not result in a conflict and is

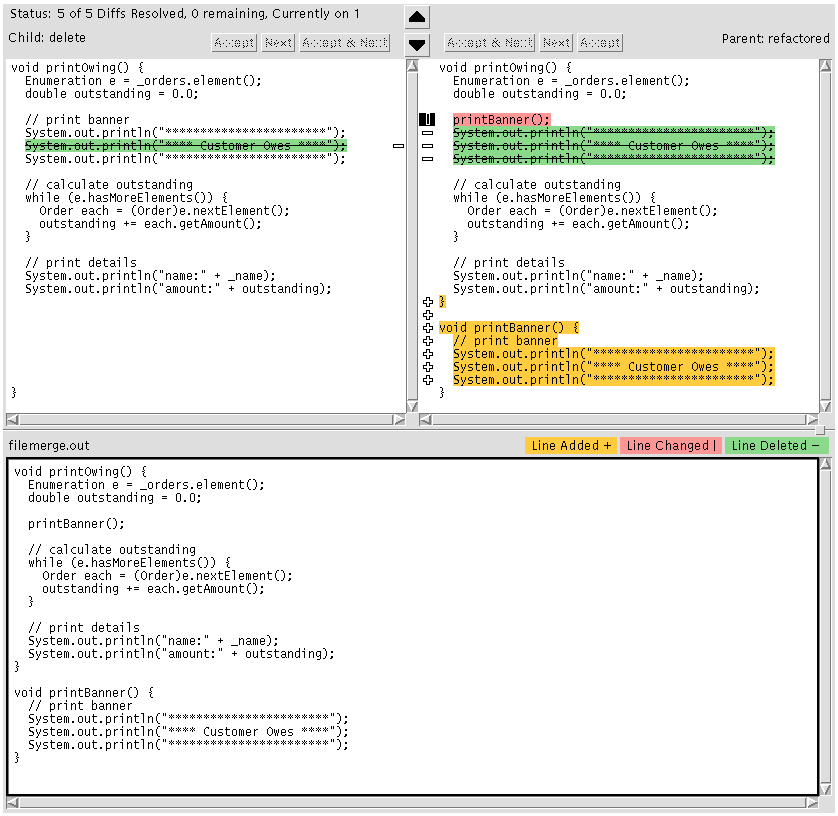

Figure 5: Extract method performed in parallel with a delete of one of the extracted lines. The merge tool doesn't detect a conflict and automatically merges the changes, but the result has the wrong semantics.

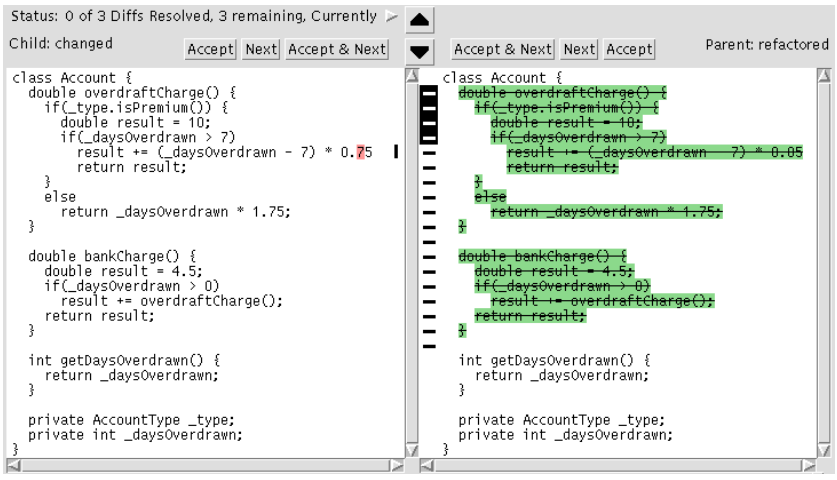

Figure 6: A change to `overdraftCharge()` is done in parallel with the move method refactoring. A conflict is detected since there has been a change and a delete on the same line. It would have been more informative to the user if the system showed that the method was moved and not, as it seems here, deleted.

automatically merged but the resulting code will not be as expected. Either it will not compile since the referenced method is no longer available or, even worse, it will have a different logical behavior if the moved method shadowed another implementation.

If the tool had static semantic information, it could detect that the added invocation referenced a method that had been moved and give a conflict. It could then provide the user with information about which class the method has moved to, how this object is reached and if the parameter format has changed.

3.3 Rename Method

When a method name doesn't reveal its purpose, change the name.

This is a quite simple but powerful refactoring; just change the name in the declaration and update all invocations to use the new name. However, in a text based system, adding new invocations in parallel to doing that refactoring would not be detected as a conflict. An automatic merge would be performed but the resulting code would not work since the added invocation tries to call the method by its old name.

If the tool knows that the changes in the first version was a *rename* and has static semantic information (name binding) it could automatically change the added invocations to the new name.

4 Merging, by intention

By adding more intelligence to the versioning/merge tool, some unnecessary conflicts can be avoided (e.g., adding two methods at the end of a class) but, even more importantly, a larger set of conflicts can be detected. That gives the programmer better support for doing the proper merge and increases awareness about the logical changes made.

4.1 Improved merging

As described above, higher order changes makes it possible to introduce the notion of moving code. This mitigates some of the shortcomings of line-based merge tools and makes it clearer where the conflict actually is.

Furthermore, if the versioning tool knows that a set of changes is a certain refactoring, the conflict reporting can be much more informative. For example, if a *move method* has been performed at the same time as we add an invocation of that method, it is much more helpful if the tool tells us to which class the method has been moved and if the parameter format has changed than just telling us that the method does not exist (and, of course, much better than getting an error at compile-time which is the typical result when using a traditional tool.)

4.2 Refactoring conflicts

In order to being able to do a refactoring, certain pre-conditions must be fulfilled. For instance, a *push down method* may only be performed if that method is used only in one subclass and *hide method* can only be done on a method not called from other objects. If a change made in parallel with the refactoring introduces

code that violates the pre-conditions of that refactoring, it is not possible to make a feasible merge. Such a merge attempt should result in a conflict.

This is a new type of conflict, where it is not the changes per se that are in conflict, but that the change is in conflict with the pre-conditions of a refactoring.

5 Related work

The ideas of grouping changes together and including semantic information to improve merging are not new. The difference between the previous work and the work presented in this paper is that in the previous work, this information is primarily used by the merge tool to do the merge or detect conflicts. We argue that this information also should be used to give better information to the programmer in order to increase awareness.

5.1 Change Oriented Versioning

Change oriented versioning[LCD+89] focuses on functional changes in a software product. It is a way of grouping a set of changes and assigning some properties to them and is intended as an extention of conditional compilation.

We use the idea of grouping changes and assigning properties to them in order to add semantics to a set of changes, for instance that they constitute a refactoring.

5.2 Structure-Oriented Merging

The IPSEN system includes structure-oriented merging[Wes91] which preserves context-free correctness and detects context-free conflicts. It also takes binding of identifiers to their declarations into account. Westfechtel also mentions that in the future, *cut and paste* operations should be taken into account. This corresponds to our high-order change *move*.

5.3 Semantic merging

The extended representation of changes proposed in this paper is limited to static semantic informantion, and our approach is to detect conflicts and give adequate information to help the programmer resolve conflicts manually. This is a big improvement over purely syntactic tools but is quite conservative in conflict detection.

There have been efforts in improving automatic merging and making conflict detection more accurate by doing full semantic analysis on the program in order to decide if two sets of changes interfere or not[HPR89]. That algorithm has been extended to detect semantics-preserving transformations[YHR92], i.e., changes to the implementation of a program that doesn't change its output. To our knowledge, the currently available systems for semantic merging are limited to handling very small imperative languages and don't support e.g. object-oriented languages. The downside of full semantic analyis is that it is highly language dependant and that it may be costly to add support for a new language.

6 Conclusions

We propose that the traditional representation of changes between revisions are extended with the notions of *higher order changes* and *static semantic information*. Higher order changes are a set of changes with a semantic meaning used to represent a logical operation, e.g., moving a piece of code or performing a refactoring. Static semantic information, e.g., name binding, is necessary in order to merge, or detect conflicts in, changes involving the declaration of entities.

The proposed extensions make conflict detection more accurate; firstly, by removing many false conflicts and, secondly, by allowing detection of semantic conflicts. It also makes it possible to report conflicts at a higher level, i.e., not only showing where the conflict is but also which type of high level change that caused it. This is particularly helpful when the scope of the changes are multiple configuration items. It also increases the programmers' awareness and guides a programmer doing a manual merge by expressing the underlying reason of the conflict.

We conclude that extending version management with higher order changes and static semantic information leads to less false conflicts, more informative conflict reporting, automatic merge of a larger set of changes and more conflicts detected by the merge tool instead of causing compile- or run-time errors.

References

[Bec99] Kent Beck. *Extreme Programming. Embracing Change.* Addison Wesley, 1999.

[Fow99] Martin Fowler. *Refactoring. Improving the design of existing code.* Addison Wesley, 1999.

[HPR89] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *ACM Trans. Prog. Lang. Syst.*, 11(3), 1989.

[JAH01] Ron Jeffries, Ann Anderson, and Chet Hendrickson. *Extreme Programming Installed.* Addison Wesley, 2001.

[LCD+89] A. Lie, R. Conradi, T. Didriksen, E. Karlsson, S. O. Hellsteinsen, and P. Holager. Change oriented versioning. In *Proceedings of the Second European Software Engineering Conference.* Springer-Verlag, 1989.

[Leo00] Alexis Leon. *A Guide to Software Configuration Management.* Artech house publishers, 2000.

[Opd92] William F. Opdyke. *Refactoring object-oriented frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Wes91] Bernhard Westfechtel. Structure-oriented merging of revisions of software documents. In *Proceedings of the Third International Workshop on Software Configuration Management.* ACM Press, 1991.

[YHR92] W. Yang, S. Horwitz, and T. Reps. A program integration algorithm that accomodates semantics-preserving transformations. *ACM Transactions on Software Engineering and Methodology*, 1(3), 1992.