Making Software Integration Really Continuous

Mário Luís Guimarães and António Rito Silva

Department of Computer Science and Engineering
IST, Technical University of Lisbon, Lisbon, Portugal
{mario.guimaraes,rito.silva}@ist.utl.pt

Abstract. The earlier merge conflicts are detected the easier it is to resolve them. A recommended practice is for developers to frequently integrate so that they detect conflicts earlier. However, manual integrations are cumbersome and disrupt programming flow, so developers commonly defer them; besides, manual integrations do not help to detect conflicts with uncommitted code of co-workers. Consequently, conflicts grow over time thus making resolution harder at late stages.

We present a solution that continuously integrates in the background uncommitted and committed changes to support automatic detection of conflicts emerging during programming. To do so, we designed a novel merge algorithm that is O(N) complex, and implemented it inside an IDE, thus promoting a metaphor of continuous merging, similar to continuous compilation. Evidence from controlled experiments shows that our solution helps developers to become aware of and resolve conflicts earlier than when they use a mainstream version control system.

Keywords: software merging, version control, continuous integration, conflict detection, continuous merging.

1 Introduction

Programming inside teams of multiple developers generally results in merge conflicts between concurrent changes. Conflicts can be difficult to detect in object-oriented programs without adequate tool support, and result in software defects as developers work more in parallel [14].

The problem with conflicts is that the later they are detected the costlier they are to resolve [1,8]. This is especially true as time passes without developers integrating their changes with those of co-workers. Not only conflicts can grow too much such that more code needs to be reworked later, but changes become less fresh in developers' minds making it more difficult to remember what was done and where to start resolution.

Recognizing this problem, good practice recommends developers to frequently integrate concurrent work to enable early detection of conflicts [1,8]. However, there are several limitations with manual integrations. First, they are disruptive because they require developers to pause their tasks, thus breaking the flow of programming. Second, they only detect conflicts with changes already committed

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 332–346, 2012.

in the Version Control System (VCS),[1] but they do not detect conflicts with uncommitted changes in the developers' working copies of the software system, so conflicts may grow as time passes thus making their resolution harder at late stages. Besides, mainstream VCSes only detect conflicts between overlapping textual regions in two versions of the same file (direct conflicts), but not between concurrent changes to different files (indirect conflicts). Third, when integration builds fail, developers still have to spend time understanding what happened and tracing failures back to the responsible changes and their authors.

This paper contributes a solution to report structural and semantic conflicts inside the IDEs of affected developers as conflicts emerge during programming. It is supported by a novel merge algorithm that continuously integrates in the background (in real-time) both uncommitted and committed changes in a team. The result is a metaphor of *continuous merging*, much like to continuous compilation inside the IDE, that alleviates developers from the burden of manual integrations for the sake of conflict detection, thus keeping them focused on programming. In contrast to our initial paper [10], this presents our background merging algorithm, and evaluates our solution using controlled experiments, showing evidence of its usefulness compared to only using a VCS.

The following sections are summarized: Section 2 describes the problem of not detecting conflicts early, and shows the limitations of current VCSes and manual integrations. Section 3 presents our solution to early conflict detection and its background merging process. Section 4 presents an empirical evaluation that sustains our solution. Section 5 lists the related work, and Section 6 concludes.

2 Problem

Imagine three developers checking out the same working copy of an application from a VCS, and then making concurrent changes. Mike ❶ changes class Mammal to extend Animal, and checks in. Anne ❷ creates class Primate by extending Mammal, adds a feature to move primates to an absolute position, merges Mike's changes from the head of the development line in the VCS, and checks in. Meanwhile, Bob ❸ changes class Animal to move animals by some distance from where they are, merges Mike's and Anne's changes, and checks in. Note that all changes were done to different files, so the merges were clean, that is, the VCS reported no conflicts. The final code in the development line is shown in Fig. 1. The problem with the final code is that an unexpected override conflict affecting the "move(int, int)" methods was not caught by the VCS, and now, it is causing this bug: if "Animal.move(int dx, int dy)" is called on a primate, this will move to position "(dx,dy)" instead of moving distance "(dx,dy)" from its current position, as expected for animals.

Using the VCS, the earliest the conflict could be found was when Bob merged Mike's and Anne's changes. Nevertheless, the VCS told him "Go ahead, the merge is clean!", so Bob has no reason to suspect Mike's and Anne's files. Even though he had written a test for "Animal.move(int dx, int dy)", this would not

[1] E.g., Subversion (`http://subversion.apache.org`) and Git (`http://git-scm.com`).

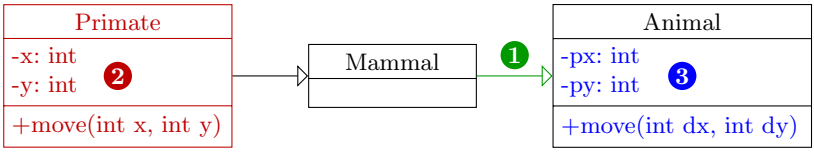


Fig. 1. The final merge at the head

check primates because Bob did not know about them when he wrote the test, so testing would not help much in this case.

The practice of frequent manual integrations has some limitations too. If Bob attempts to frequently integrate the code of his colleagues, he will probably bring code tangential to his work [1], and interrupt him too much. Because manual integration does not detect conflicts with uncommitted code in working copies, the best chance to detect the conflict is if Mike and Anne checked in often. However, they may prefer to defer check-ins until when they are finished with their tasks, thus delaying the detection of the conflict. On the other hand, this practice requires developers to check-in partial changes just for the sake of conflict detection, thus causing distraction and polluting the VCS with insignificant check-ins.

Eventually, a few days later a user approaches Bob: "Do you remember that animal move feature I asked?", "Yes!?", "It does not work for gorillas!", "How's that?", "Well, you coded it, go figure out!". Unfortunately, time passed and changes are no longer fresh in Bob's head, so he will have to work harder to investigate and resolve the bug. He will have to remember what he did before, determine the impact of the bug on other parts of the code, approach his colleagues if they are available, and decide what to do. At least he will have to remove one of the duplicated points, rename one of the "move" methods, and change where in the code there are dependencies on the removed point and the renamed method. All this requires more time and effort than if the conflict was detected earlier. This example is simple but shows that conflicts can be difficult to detect and are costly to resolve when found late.

Wouldn't it be helpful a tool that did continuous (real-time) integration in the background to automatically detect the above conflict as it emerged during programming, and reported it inside the IDE, thus exempting developers from manual integrations and from all that rework? This is what our solution does.

3 Solution

Our solution assumes that a software project comprises one or more teams of developers, each team working along a *development line* (or *branch*) [1] supported by a mainstream VCS, as shown in Fig. 2.

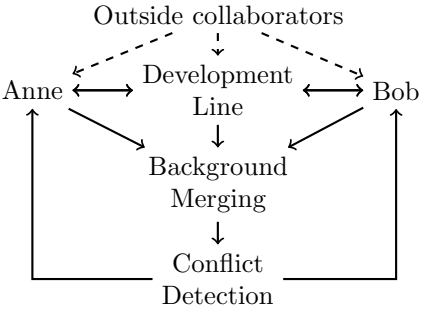


Fig. 2. The information flow inside a team

Team members follow the typical "copy-modify-merge" process: they check out working copies of the system from the development line, modify the working copies, merge other members' and outsiders' changes into the working copies, directly or via the line, and check in their working copies into the line.

Simultaneously, changes to the code are captured when files are saved in working copies or checked in into the development line, and transmitted to background merging in order to continuously update a background system, called the team's *merged system*. This system is then post-processed to automatically detect conflicts as they emerge inside the team. In addition, when members leave the team the effect of their changes is removed from the merged system.

Changes in working copies are sent to background merging automatically, or manually if the developer wants to take control — for example, a developer may decide to transmit only when changes are reasonably stable. Anyway, changes can only be sent if the working copy compiles successfully, to avoid syntactically invalid code entering the merged system. On the other hand, changes at the head of the development line are always automatically processed, so developers should guarantee that check-ins do not carry compilation errors into the development line, which is a good practice and an easy one to ensure using today's IDEs and VCSes (e.g., via pre-commit hooks).

Conflicts are reported in detail to affected members inside a view in the IDE, much like compilation errors are reported today, thus promoting a metaphor of *continuous merging*.

3.1 Tracking Changes

The working copies, the check-ins along the development line, and the merged system are abstractly modeled as trees of labeled, typed, and attributed nodes representing the physical folders, files, and program elements. The labels allow to match nodes in different trees for computing changes between consecutive trees, and the types and attributes define how source code is stored in trees. Labels, types, and attributes are specific to the language domain.

We implemented our solution for Java programming, and chose the labels as the names of folders, files, fields, and the signatures of methods. In some cases,

to disambiguate program elements in the same scope, like classes and interfaces, the labels are the concatenation of name and type. Only folder nodes do not require attributes.

Fig. 3 exemplifies how changes inside the team are tracked and merged in the background (this figure will guide us throughout Section 3). In Fig. 3a, Anne and Bob made several concurrent changes to base file F.java, which were merged in the background into the file in the merged system (background merging is described in Section 3.2). In Fig. 3b, the content of some nodes in the base file is shown as an example of how source code is mapped to nodes and attributes.

Fig. 3c shows the trees of the base file and those of Anne's and Bob's working copies of that file.[2] As developers change the code, the working copy tree evolves from the base tree as follows: unchanged nodes are shared by the two trees (e.g., Bob did not change node "e"); added nodes only exist in the working copy tree (e.g., Anne added node "a"); deleted nodes only exist in the base tree (e.g., Anne deleted node "e"); and changed nodes appear as new nodes in the working copy tree (e.g., Anne's and Bob's node "pi"). In addition, added, changed, and deleted nodes cause a change of their parent nodes (*change propagation*), as it happened with file node "F.java". The arrows represent the succession relationships between the nodes in consecutive trees (the successor points to the predecessor). Likewise, the evolution of nodes between the trees of consecutive check-ins in the development line follows the same rules.

The Evolution Tree. The evolution of the software as it changes is tracked in the evolution tree, shown in Fig. 3d for F.java. The entries in this tree are called **evolution graphs**, and they capture the succession relationships of the nodes in the same labeled position in the working copies and the check-ins along the development line, as shown by the arrows. In a graph, the dark circles represent added or changed nodes in a tree, and the white circles, called **null nodes**, represent deleted nodes in a tree. In the figure, the labels "b", "A", and "B", indicate from which trees in our example the nodes come from (e.g., graph "e" shows a node deleted by Anne that is shared between the base and Bob). A node is **older than** another if it precedes the other on the transitive closure of the succession relationships (e.g., Bob's node "e" is older than Anne's null node "e").

In a graph, the nodes that are not succeeded are called **forefront nodes** because they contain the most recent edits to the attributes. A graph is n-way if its number of forefront nodes is n (> 0). A graph is **consistent** if all forefront nodes, except null nodes, have the same type (in this case we say that the graph is "a consistent <type>" or is "of type <type>"); otherwise, it is **inconsistent** — this is the bizarre case of one developer adding a file while another developer adds a folder with the same name. For Java, inconsistent nodes only occur at folder and file level. A graph is a **null graph** if it has only null nodes at the forefront.

The evolution tree is used to update the merged system during background merging and to identify members affected by conflicts.

[2] We omit parent folders for clearness, but the discussion applies to folders too.

F.java (base)

```
class F {
 float e = 2.7f;
 public float pi;
 int q = 1;
 int m(){
  return 1;
 }
}
```

F.java (Anne)

```
final class F {
 float e = 2.7f;
 public float pi =3.14;
 int a = 1;
 int q = 2;
 int m(){
  return q;
 }
}
```

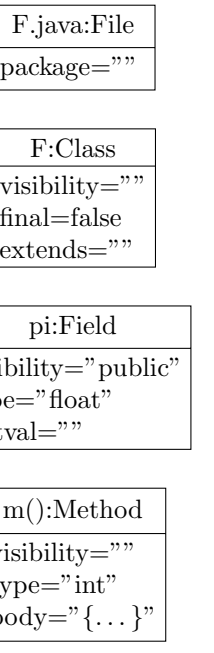
F.java (Bob)

```
public class F {
 float e = 2.7f;
 public float pi;
 int b = 2;
 int q = 3;
 int m(){
  return 1;
 }
}
```

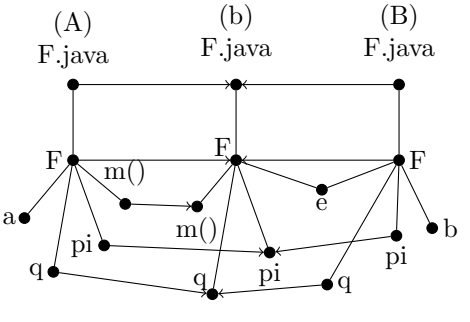
F.java (merged system)

```
public final class F {
 float pi=3.14;
 int a = 1;
 int b = 2;
 int q = 0;
 int m(){
  return q;
 }
}
```

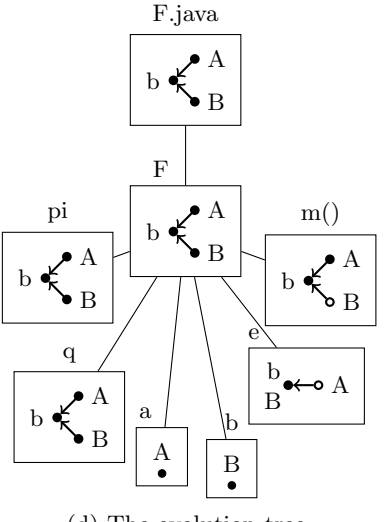
(a) Anne's and Bob's changes to F.java (indicated by the rectangles and the strikes), and the resulting file in the merged system.



(b) Some nodes in F.java (base).



(c) The trees of (b)ase, (A)nne, and (B)ob.



(d) The evolution tree.

Fig. 3. Tracking changes inside the team

3.2 Background Merging

Background merging uses the evolution tree to do *automatic and incremental n-way structural merging* of the most recent changes to the software. It is:

- automatic because all structural conflicts are temporarily resolved in the merged system using *default resolutions* to not stall background merging. One example is the *deletions rule*, which ignores forefront null nodes when other forefront nodes exist, in order to favor changes over deletions;[3]
- incremental because it has to "remerge" only the folders and files that have been modified in the working copies or in the check-ins since the last call to background merging;
- and n-way because it merges the forefront nodes in each graph in one pass.

Initialization. The merged system is initialized with the tree at the development line's head, i.e., $MS = H$. Then, for each call i, MS is updated as follows:

Folder Merging. Visit bottom-up the subtree comprising the graphs in the evolution tree corresponding to the folders and files that were modified after call $i - 1$,[4] and follow these rules at each visited graph:

- If the graph is inconsistent, delete the corresponding node in MS.
- If the graph is a consistent file, proceed to "File Merging" (see below).
- If the graph is a consistent folder, create that folder in MS if it does not exist already there (e.g., it exists before call i, or it was created during call i because some child was created).
- If the graph is null and the corresponding node in MS is a file, or a folder having no children, delete that node in MS.

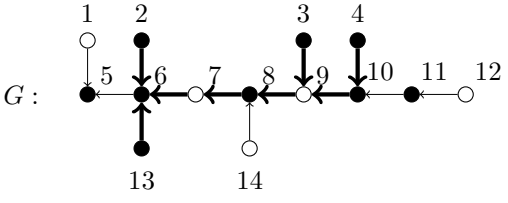
File Merging. Run these ordered steps at each visited graph G (top-down), starting with the evolution graph corresponding to the file:

1. Identify the graph G' to be merged. Let $G' = G$, then modify G' following the order of these steps:
   (a) Ignore all null nodes at the forefront whose parent nodes are null and were ignored in the preceding visited graph. This step is explained with an example. When a developer deletes a class and another developer changes one of its methods, MS should include the entire class with the method change to avoid programming language inconsistencies (e.g, the change may use fields and methods that were deleted with the class). Therefore, this step ignores all null nodes corresponding to the class and its descendents that were deleted, so the class is not incomplete in MS.

[3] We have tested this decision with twenty-one graduate students by exposing them to a "change & deletion" situation, and they all decided to preserve the change.

[4] Bottom-up traversal supports files checked out from different points in the development line, and checkouts of partial trees.

(b) For the remaining nodes, let N = {all null nodes at the forefront} and $\overline{N}$ = {all non-null nodes at the forefront}. Then, if $\overline{N}$ is not empty (you may follow the example in the figure below in which the final G' corresponds to the nodes linked by the strong arrows):
   i. Ignore all nodes in N (*deletions rule*);
   ii. Ignore all nodes succeeded by those in N that are not succeeded by those in $\overline{N}$ (nodes only succeeded by ignored null nodes do not contribute with attribute edits to the merge);
   iii. Ignore all nodes succeeded by the oldest node that directly precedes one in $\overline{N}$ (the oldest or its successors contribute to the values of all attributes of the forefront nodes in G', so we can cut the "tail").



Ignored at each step:

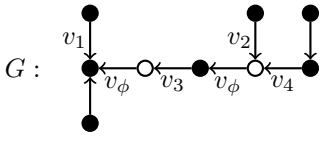
i. $\{1, 12, 14\}$
ii. $\{11\}$
iii. $\{5\}$

2. Now do the one that applies:
   – If G' is a null graph, delete the corresponding node in MS.
   – If G' has only one node at the forefront, copy that node's tree to MS.
   – If G' has several nodes at the forefront:
      (a) Copy the merge of G' to MS as per "Merging a Graph" (see below).
      (b) Repeat 1, now for each of the evolution graphs corresponding to the children of the nodes at the forefront of G'.

Applying these steps to merge Anne's and Bob's modifications to base file F.java will result in "F.java (merged system)" listed in Fig. 3a.

Merging a Graph. A consistent graph G of type T is merged as follows:

1. Set $n \leftarrow$ a new node of type T.
2. For each n_a attribute of n do:
   (a) Set $E \leftarrow$ {the values of concurrent edits of attribute a in G}. For example, given the values of the edits of a (the v_is, and the v_ϕs of null nodes) in the next figure, $E = \{v_1, v_2, v_4\}$:



   (b) If $\#E = 0$: Set $n_a \leftarrow$ the single value of a in all forefront nodes (because there are no concurrent attribute edits we maintain the former value).
   (c) If $\#E = 1$: Set $n_a \leftarrow$ the single value in E (either there is a single edit, or all concurrent edits set the same value).

(d) Otherwise ($\#E > 1$): Set $n_a \leftarrow$ the default value for a, to not stall background merging.

3. Return the merge node n.

Complexity Analysis. Performance of background merging is assessed via a cursory complexity analysis.

Time Complexity. File merging is $O(N_{elem} \times R_{max} \times A_{max})$ where N_{elem} is the number of elements visited in a file, R_{max} is the maximum number of nodes in any of those elements' graphs, and A_{max} is the maximum number of attributes in any of those elements' types. Since A_{max} is bounded by the programming language, file merging is $O(N_{elem} \times R_{max})$. Background merging updates in the worst case N_{files} and calls file merging for every file, so its time complexity is $O(N_{files} \times N_{elem} \times R_{max}) \approx O(N \times R_{max})$, where N is the total number of elements to "remerge". In practice, R_{max} is bounded by the maximum team size, which is generally small for teams in a project to be manageable, so the overall time complexity is $O(N)$. In our implementation, we merged 16 file versions (484 LOC each on average) in $4.3ms$.

Space Complexity. This is proportional to the memory needed to maintain the evolution tree, so in the worst case it is $O(N_{system} \times R_{max})$, where N_{system} is the total number of folders, files, and elements in the software system being created by the team, and R_{max} is the same as above. Like before, this can be approximated by $O(N_{system})$ in practice. Besides, the evolution tree is computed and stored in memory as needed.

3.3 Conflict Detection

Structural Conflicts. These are detected during background merging. They are temporarily resolved in the merged system using *default resolutions*, defined for the language domain. Default resolutions are necessary to not stall background merging, yet these conflicts persist in the affected working copies until the team resolves them after being informed. The types of structural conflicts are these:

- *pseudo direct conflict*: occurs when different attributes of a node are concurrently changed, or the same attribute is concurrently changed to the same value. It is a warning that reminds of a possible semantic conflict at language level (see below). In Fig. 3a, it happened with class "F" and field "pi";
- *attribute change & change conflict*: occurs when the same attribute of a node is concurrently changed to different values. The default resolution is to assign a default value to the attribute in the merged system's node according to the attribute's type. In Fig. 3a, this conflict was temporarily resolved (gray color) for the "initval" attribute of field "q" by setting it to zero (the default value of "int");

- *node change & deletion conflict*: occurs when there are concurrent changes and deletions to the same node, and the default resolution is to apply the deletions rule. In Fig. 3a, it happened with method "m()", so Anne's change prevails in the merged system;
- *inconsistent graph conflict*: occurs when an evolution graph becomes inconsistent, and the default resolution is to delete the corresponding node in the merged system. This weird case should never happen, yet we handle it.

Note that developers are always alerted to structural conflicts, and once they resolve them in their working copies the merged system is updated with their resolution in the next iteration of background merging. This is why default resolutions in the merged system are always temporary.

The remaining conflicts are detected by post-processing the merged system.

Language Conflicts. The merged system is immediately compiled after being updated. As explained before, all changes only enter background merging if they are syntactically valid, so compilation errors in the merged system can only result from invalid combinations of concurrent changes with respect to the static semantics of the programming language.

As such, post-processing listens the compilation output, processes the errors, and reports them as *language conflicts* back to the IDEs of affected members. This is a very effective solution because it avoids to re-implement complex programming language rules. One case is the *undefined constructor conflict*, which occurs when one developer adds a constructor with one argument to a class having no constructors, while another developer creates a subclass of that class.

Behavior Conflicts. These represent undesired behavior because of unexpected interactions between merged changes. They are detected by searching for *conflict patterns*, that is, logical conjunctions of facts regarding the program elements and their semantic dependencies in the merged system that identify potentially unwanted behavior.

The more specialized patterns are the more interesting ones because the conflicts they represent are hard for developers to find without tool support. This is the case of the *unexpected override conflict* in Section 2, which is found using the pattern $\exists A, B, m_1, m_2 \in G : extends*(A, B) \wedge method(A, m_1) \wedge method(B, m_2) \wedge equalSignature(m_1, m_2)$, where A is a super class of B and $extends*$ is the transitive closure of the *extends* dependency.

Note that conflicts only occur if the instantiated facts correspond to nodes changed by different members (we omitted this part in the example pattern to avoid complicating it). An advantage of using conflict patterns is that they can be easily added to support more behavior conflicts.

Test Conflicts. These are detected by running automated tests in the merged system. A test conflict is one that fails and its execution flow has reached methods changed by different members. Suppose that Anne adds a test to verify that all species have a price defined by method "getPrice()", and Bob adds class

Chimpanzee without such method because he is not aware of Anne's new feature. As such, Anne's test will fail in the merged system when retrieving the price of Chimpanzee, like this execution flow shows:

```
zoo.testing.ZooTests.setUp()✔
zoo.testing.ZooTests.testAnimalGetPrice()✔                    (Anne)
...
zoo.animals.Animal.getPrice(Ljava/lang/Class;)✔
zoo.animals.Chimpanzee.getPrice()✖                            (Bob)
```

A test conflict is detected for "testAnimalGetPrice()" because its execution reached Anne's new method (the test) and tried to call "getPrice()" on Bob's new class via reflection. Post-processing places hooks in the reflection API (via bytecode instrumentation), and checks if missing methods were deleted or never existed, which was the case for Bob, thus detecting a *missing method conflict*.

3.4 Reporting Conflicts

A conflict is reported to the members that changed the nodes affected by it. These are the nodes that were "remerged" (structural conflicts), the nodes involved in a compilation error (language conflicts), the nodes that instantiate the facts of a conflict pattern (behavior conflicts), and the nodes corresponding to the methods in failed execution flows (test conflicts). Only the members that modified these nodes will receive notifications for the conflicts affecting the nodes. The evolution tree tracks who modified which nodes, so to find these members we look for those nodes in this tree.

4 Evaluation

Our evaluation shows that developers using our solution become aware of and resolve conflicts earlier than when they use only their VCS. It was done via controlled user experiments, as described next.

The Tool. In order to evaluate, we implemented WECODE as an extension to the Eclipse IDE (`http://www.eclipse.org`). The main screen of WECODE is shown in Fig. 4. The Team view ❸ shows all members in the team and the details of their changes down to program elements. It shows yellow and red icons to respectively signal pseudo or more urgent structural conflicts on folders and files. If they wish to control change transmission, developers can manually transmit changes to background merging only at stable moments of their tasks. Developers can also update their code with other members' changes in order to early resolve conflicts while changes are vivid in their minds (a chat view, not shown, facilitates the discussion of changes and resolutions). The Team Merge view ❹ reports semantic conflicts: notifications have detailed messages

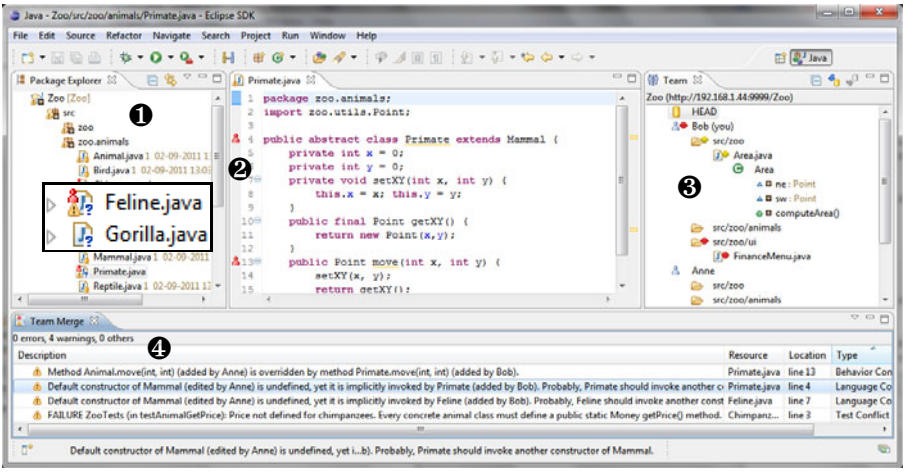


Fig. 4. Continuous merging inside the IDE

describing the conflict, the affected program elements, the affected members, and how elements were changed. This fosters conflict resolution. In addition, conflicts are signaled at affected files ❶ and program elements ❷.

Controlled Experiments. *Does our tool help developers become aware of and resolve conflicts earlier than when they use a mainstream VCS?* To answer this, we organized two groups of 7 graduate software engineering students: the WECODE and the VCS groups. Each subject teamed with a confederate, who inserted the same conflicts (those in the Team Merge view ❹) at equivalent times for all subjects, before half of their tasks were done. The application (41 classes, 1143 LOC) and the tasks were designed by the authors, and the application was sent to all subjects at least two days before their experiment. Before start, the subjects watched a tutorial video of the tools to use (WECODE and Subversion). At the end, they were asked to check in and resolve any remaining conflict.

Results. WECODE subjects became aware of all 28 conflicts (7 subjects × 4 conflicts) as they emerged during their tasks before check in, whereas VCS subjects detected no conflict before check in (these are statistically significant results by Fisher's or Pearson χ^2's tests of the corresponding 2x2 contingency table). At check in, VCS subjects only detected the language conflicts because of the compilation errors (they forgot to run the tests so they missed the test conflict). Since there were no direct conflicts VCS subjects did not pay attention to the files changed by the confederate. Only one VCS subject was observed to frequently integrate with the VCS after each task, but there was no direct conflicts so he said "there is no problem here".

Regarding the WECODE group, subjects resolved conflicts early and generally between their tasks. The average delay to start resolving each conflict type was (we removed other conflicts' resolution time from delays): language conf. (4m22s, sd=5m1s); behavior conf. (2m10s, sd=2m44), and test conf. (3m1s, sd=3m55). This was also the order that conflicts emerged, and even though the sample was small, it is interesting to observe that the delay slightly decreased as subjects got more used with the tool.

Both groups were asked to score (from 1 to 10) if they (liked / would like) to be informed about conflicts during programming, instead of only at check-in. WECODE subjects scored 9.3 (sd=0.70) and VCS subjects scored 8.00 (sd=1.85), thus showing a high desire for such feature. Asked why, they said:

> VCS: "I would not have to read many conflicts at the end when I probably had forgotten the changes I made" and "It would make development more interactive, and I would anticipate when a colleague is breaking our code"
> MERGE: "Because it informs me in useful time, thus sparing me time looking for errors later. All time wasted tracing error messages in the build would be spent doing useful things"

About our tool, the subjects said "I liked the icon in the editor informing me about the conflict and with whom" and "I liked most its simplicity of use".

Threats to Validity. Subjects did not knew they would be evaluating our tool to not influence their behavior and responses. They were randomly selected into the groups, and all had experience with the VCS. Using Subversion or another mainstream VCS would not change the results: all they do is textual merging. The code to type in every task was given to them thus eliminating the effect of different programming skills. Regarding external validity, our conflicts might be threatened regarding their occurrence in practice. Studies are needed to understand the nature of conflicts, however it is reasonable to assume that conflicts in real projects are at least as difficult to detect as those we chose. Our results indicate that continuous merging can be beneficial, still we believe that experimenting with real projects will provide further insight into our work.

5 Related Work

Our work relates to others in the areas of software merging and awareness.

Software Merging. Textual merging (Unix's diff3, and all mainstream VCSes) blocks when textual conflicts occur, like when different attributes are changed on the same line of text, and fails to match changes when the program elements are reordered inside concurrent files. Consequently, background textual merging, like done in [4], has these limitations. In contrast, our background structural merging never blocks, handles changes to different nodes and attributes transparently, and supports reorderings, so it detects more important conflicts (semantic) earlier. *Flexible structural merging* [13] merges two versions of a file

using two-dimensional matrices that decide how merging is done at node level (manual or automatic). Flexibility is achieved by configuring matrices for different collaboration scenarios. This solution does not scale for more than two versions of a file, because matrices need to be reconfigured each time the number of versions varies, which is unfeasible in practice. *Semantics-based merging* [3] has been mostly theoretical achievements using very limited languages. *Operation-based merging* [12] serializes two concurrent sequences of operations. Tools must capture all changes as operations, but the editors developers use are not operation-based. Sequences may grow too much, and redundant operations must be eliminated to avoid false conflicts. In contrast, our solution adapts to the tools developers use.

Awareness. Solutions based on *awareness* [7] report which files, types, and program elements are being changed at the moment by co-workers, which may help to detect conflicts early. These solutions may overload developers with notifications that are irrelevant to what they are doing [5,9,11], and require developers to investigate the notifications to determine if they bear any conflict. This can be difficult because of the complex semantic dependencies between program elements (e.g., polymorphism and late binding). Besides, this steals time from programming. For example, some solutions report direct conflicts even when changes are done to independent program elements in a file [2,15]. A structure-based solution like ours does not have this shortcoming. Others go beyond direct conflicts, and notify when two files, types, or program elements, connected by a path of semantic dependencies, have been concurrently changed by the developer and a co-worker [6,16,17]. Nevertheless, in these solutions, developers still have to investigate the notifications to identify where real conflicts exist.

6 Conclusion and Future Work

Early detection of conflicts is important to facilitate resolution. The recommended practice is to frequently manually integrate others' changes, but this is too much burden and disrupts the flow of programming. In contrast, we presented a solution that does really continuous integration in the background in order to automatically detect conflicts as they emerge during programming, and reports them in detail inside the IDE. An empirical evaluation demonstrated that our solution makes developers aware of conflicts that are difficult for them to find using current tools, and fosters early resolution while changes are still fresh. This support clearly contrasts with the tools developers use today.

Our research will proceed in several directions. We want to support refactorings and domains beyond programming, like collaborative model-driven engineering. We want to evaluate our solution via a longitudinal study with professional programmers in order to adjust it to real projects, and to understand how continuous merging influences their software process, for example, if new collaboration patterns emerge. In the long term, we want to measure the overall effect of continuous merging on software quality.

References

1. Berczuk, S., Appleton, B.: Software Configuration Management Patterns: Effective Teamwork, Practical Integration. Addison-Wesley, Boston (2002)
2. Biehl, J., et al.: FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams. In: ACM SIGCHI Conf. on Human Factors in Computing Systems, CHI 2007, pp. 1313–1322. ACM Press, New York (2007)
3. Binkley, D., et al.: Program Integration for Languages with Procedure Calls. ACM Trans. Softw. Eng. Methodol. 4(1), 3–35 (1995)
4. Brun, Y., et al.: Proactive Detection of Collaboration Conflicts. In: 8th Joint Meet. of the Euro. Softw. Eng. Conf. and ACM SIGSOFT Symp. on the Foundations of Softw. Eng., ESEC/FSE 2011, pp. 168–178. ACM, New York (2011)
5. Damian, D., et al.: Awareness in the Wild: Why Communication Breakdowns Occur. In: Inter. Conf. on Global Softw. Eng., ICGSE 2007, pp. 81–90. IEEE Computer Society, Washington, DC (2007)
6. Dewan, P., Hegde, R.: Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development. In: Bannon, L., Wagner, I., Gutwin, C., Harper, R., Schmidt, K. (eds.) ECSCW 2007, pp. 159–178. Springer, London (2007)
7. Dourish, P., Bellotti, V.: Awareness and Coordination in Shared Workspaces. In: ACM Conf. on Computer Supported Cooperative Work, CSCW 1992, pp. 107–114. ACM, New York (1992)
8. Fowler, M.: `http://martinfowler.com/articles/continuousIntegration.html`
9. Fussell, S., et al.: Coordination, Overload and Team Performance: Effects of Team Communication Strategies. In: ACM Conf. on Computer Supported Cooperative Work, CSCW 1998, pp. 275–284. ACM, New York (1998)
10. Guimarães, M., Rito-Silva, A.: Towards Real-Time Integration. In: 3rd Inter. Workshop on Cooperative and Human Aspects of Softw. Eng., CHASE 2010, pp. 56–63. ACM, New York (2010)
11. Kim, M.: An Exploratory Study of Awareness Interests about Software Modifications. In: 4th Inter. Workshop on Cooperative and Human Aspects of Softw. Eng., CHASE 2011, pp. 80–83. ACM, New York (2011)
12. Lippe, E., van Oosterom, N.: Operation-based Merging. In: 5th ACM SIGSOFT Symp. on Softw. Dev. Environ., SDE 5, pp. 78–87. ACM, New York (1992)
13. Munson, J., Dewan, P.: A Flexible Object Merging Framework. In: ACM Conf. on Computer Supported Cooperative Work, CSCW 1994, pp. 231–242. ACM, New York (1994)
14. Perry, D., et al.: Parallel Changes in Large-Scale Software Development: An Observational Case Study. ACM Trans. Softw. Eng. Methodol. 10(3), 308–337 (2001)
15. Sarma, A., et al.: Palantír: Raising Awareness among Configuration Management Workspaces. In: 25th Inter. Conf. on Softw. Eng., ICSE 2003, pp. 444–454. IEEE Computer Society, Washington, DC (2003)
16. Sarma, A., et al.: Towards Supporting Awareness of Indirect Conflicts Across Software Configuration Management Workspaces. In: 22nd IEEE/ACM Inter. Conf. on Aut. Softw. Eng., ASE 2007, pp. 94–103. ACM, New York (2007)
17. Schümmer, T., Haake, J.: Supporting Distributed Software Development by Modes of Collaboration. In: 7th Euro. Conf. on Computer Supported Cooperative Work., ECSCW 2001, pp. 79–98. Kluwer Academic Publishers, Norwell (2001)