

# Improving Early Detection of Software Merge Conflicts

Mário Luís Guimarães and António Rito Silva  
Department of Computer Science and Engineering  
INESC-ID, IST, Technical University of Lisbon  
Lisbon, Portugal  
{mario.guimaraes, rito.silva}@ist.utl.pt

**Abstract**—Merge conflicts cause software defects which if detected late may require expensive resolution. This is especially true when developers work too long without integrating concurrent changes, which in practice is common as integration generally occurs at check-in. Awareness of others’ activities was proposed to help developers detect conflicts earlier. However, it requires developers to detect conflicts by themselves and may overload them with notifications, thus making detection harder.

This paper presents a novel solution that continuously merges uncommitted and committed changes to create a background system that is analyzed, compiled, and tested to precisely and accurately detect conflicts on behalf of developers, before check-in. An empirical study confirms that our solution avoids overloading developers and improves early detection of conflicts over existing approaches. Similarly to what happened with continuous compilation, this introduces the case for continuous merging inside the IDE.

**Keywords**—version control; merge conflicts; awareness; continuous merging

## I. INTRODUCTION

Because of productivity, today almost all software systems are built by teams of programmers working in parallel. In this context, software merging concerns the creation of a version of the system that integrates the intentions of concurrent changes.

In general, developers modify private working copies of the system to ensure stability during programming. However, this prevents them from knowing what co-workers are doing which may affect private work. Therefore, conflicts emerge due to concurrent work, and become more complex as changes grow without being integrated and as further developments are made. Consequently, the later conflicts are detected, the harder it is to resolve them because more code must be reworked. Besides, a conflict detected late is generally harder to resolve since the changes that caused it are no longer fresh in developers’ minds [1], [2].

In a field study, Perry et al. [3] concluded that the number of software defects increase with parallel work, which was found to be considerable and inadequately supported by tools. Their conclusions remain valid today as parallel work increases with the growing distribution of software teams, which still use tools and processes similar to those found in their study.

In a recent work, Brun et al. [4] studied nine of the most active open source projects in GitHub (<http://github.com>), and concluded that even with modern version control systems, like Git (<http://git-scm.com>), merge conflicts are “frequent, persistent, and appear not only as overlapping textual edits but also as subsequent build and test failures”.

Industry experts have proposed several best practices to control merge conflicts, like Continuous Integration [2], [5], which recommends frequent merges and check-ins to avoid conflicts staying undetected for too long. Unfortunately, merging is cumbersome and disrupts programming flow, so some developers do not merge as frequently as desirable — teams avoid parallel work because of difficult merges [6], and developers rush their tasks to avoid being the ones responsible for the merge [7].

To support developers, *awareness* [8] of co-workers’ activities helps break the isolation of private work by informing developers where in the code their co-workers are currently making changes. This information can be used by the developer to detect conflicts earlier. However, because of the complex semantics of today’s programming languages, like polymorphism and late binding, it is very hard for developers to detect conflicts by themselves while they are programming. In addition, awareness may overload developers with too much information, thus making detection of conflicts harder [9], [10], [11].

This paper presents a novel solution that reduces the amount of information developers have to digest. The present solution continuously merges uncommitted and committed changes to create a background system that is analyzed, compiled, and tested to detect conflicts with high precision and accuracy on behalf of developers, while they are programming, that is, before check-in. Detected conflicts are then presented to the affected developers inside the IDE. In comparison to our initial paper [12], this details the evolution of our solution, presents our full-fledged tool, and its empirical evaluation using controlled user experiments.

The contributions of this paper are the following:

- a novel solution that introduces *continuous merging* inside the IDE, much like to continuous compilation;
- an empirical evaluation which provides quantitative and qualitative evidence that our solution improves early detection of conflicts when compared with existing

approaches based on change and dependency-based awareness.

Subsequent sections are summarized as follows. Section II motivates the need to detect conflicts early. Section III explains the limitations of existing awareness approaches to this problem, and Section IV describes our solution. Section V presents the implementation of our solution in Eclipse, and Section VI reports an empirical evaluation that sustains our solution. Section VII lists the related work, and Section VIII concludes.

## II. MOTIVATION

The need to detect conflicts early was determined by industry and research and it is illustrated next.

Suppose that Mike, Anne, and Bob check out the same working copy of the Zoo application from their mainstream Version Control System (VCS). Mike ❶ changes class Mammal to extend Animal, and commits. Simultaneously, Anne ❷ creates class Primate by extending Mammal, and adds “Primate.move(int x, int y)” to move primates to location “(x,y)”. Then, she does a clean merge with Mike’s changes at the head of the repository, and commits. Meanwhile, Bob ❸ adds method “Animal.move(int dx, int dy)” to move animals by some distance from their current location, updates his working copy with Mike’s and Anne’s changes at the head, and finally commits. Note that all VCS merges were clean because the developers changed different files, though a merge conflict exists in the final state at the head, shown in Fig. 1.

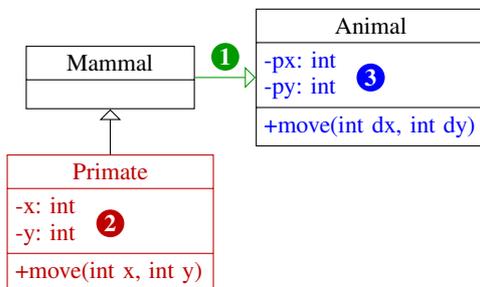


Figure 1. The final merge at the head.

The merge conflict is an unexpected override between the methods added by Anne and Bob, and at runtime it will cause the following bug: when method “Animal.move(int dx, int dy)” is invoked on a primate, this will move to location “(dx,dy)” instead of moving distance “(dx,dy)” from its current location, as expected for all animals.

Note that this conflict is very difficult to find. Only Bob could detect it since after merging Mike’s and Anne’s changes from the head only his working copy has the conflict. Nevertheless, the VCS told him that the merge was “clean”, so he does not suspect his co-workers’ changes. Besides, he had to merge other files, making him overlook

those of Mike and Anne. He even had a test for Animal, which ran successfully before he checked in. However, it did not consider primates because Bob did not know about that class when he wrote the test. Or all he wanted was to rush his check-in. Unfortunately, the bug will enter production, and further developments will be made on top of broken code.

Eventually, the bug will be found and the developers will have to resolve it. They will have to remember what they did before, determine the impact of the bug on other parts of the code, and decide what to do. All this certainly takes time because the changes are no longer fresh in their minds. At least they 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 renamed method. Although simple, this example shows that conflicts can be difficult to detect and are costly to resolve when found late.

## III. PROBLEM

Would it not be helpful to detect the above conflict as it emerges during programming and avoid all that rework?

Awareness, defined as “an understanding of the activities of others, which provides a context for your own activity” [8], has been suggested to help developers detect conflicts early [13], [14], [15], [16]. In general, these proposals use presence and change awareness.

Presence awareness informs where others are looking in the code [13], which may be useful to find co-workers for collaboration, though it does not help with conflict detection.

Change awareness informs where changes are being made in the code, which may help detect conflicts early. However, reporting which files, types, or program elements are being changed may overload developers with notifications that are irrelevant to what they are doing [9], [10], [11]. Some of these tools use *dependency-based awareness* [13], [14], [15], and notify when two files, types, or program elements, connected by a path of dependencies (e.g., “extends” and “calls”) with length  $n \geq 0$ , have been simultaneously changed by the developer and a co-worker. Their idea is to reduce the number of notifications to help developers focus on the most relevant ones, which may indicate a conflict with co-workers. For example, instead of notifying concurrent changes in every file, some tools only notify concurrent changes to the dependencies of files changed by the developer.

Notwithstanding, dependency-based awareness does not prevent developers from being overloaded, especially when the choice of granularity in the tool is that of file or type level — for example, it causes false direct conflicts ( $n = 0$ ) and false indirect conflicts ( $n > 0$ ) because of concurrent changes to independent program elements in the same type or dependent types, respectively. In addition, developers still have to investigate the notifications to determine if they

bear any conflict, thus stealing time from programming. Unfortunately, it can be difficult for developers to detect conflicts by themselves because of the complex interdependencies between program elements, like polymorphism and late binding. For example, in Fig. 1, it is unlikely that Mike upon receiving the notifications “(Primate, Mammal)” and “(Mammal, Animal)” would be able to correlate them to discover an unexpected override, especially if he is busy and tired of irrelevant notifications.

#### IV. SOLUTION

Our solution continuously merges all uncommitted and committed changes inside a software team to create a background system that is analyzed, compiled, and tested to precisely and accurately detect conflicts on behalf of developers before check-in. This introduces the case for continuous merging inside the IDE, similarly to the current experience of continuous compilation.

##### A. Collaborating inside Teams

The model of our solution is that of a team of programmers working along a *development line* (or *branch*) [1] in a VCS, as shown in Fig. 2.

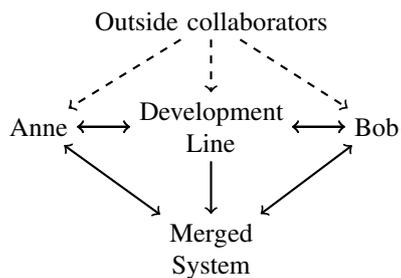


Figure 2. The Team Model.

A *team* comprises a set of members (e.g., Anne and Bob), a development line, and a special system that merges all uncommitted changes of the members and those committed in the development line, called the *merged system*. In addition, the team’s development line and its members may receive code from the outside, that is, from other programmers or teams participating in a major joint development. (Code exchanges with outside collaborators go in both directions. However, we are only interested in those incoming to the team. They also follow project-specific policies that are not relevant to discuss in this paper.)

The first member creates the team and associates it with the development line. This initializes the merged system with a copy of the system at the head (of the development line). At any time, members may join and leave the team, which continues to exist until the last member leaves.

Members modify their working copies of the system in the development line using common VCS operations — check out, modify, update, merge, check in, pull, and

push. This “copy-modify-merge” process generates a flow of changes, from all members and the head, which are then sent to update the merged system in the background. Changes in working copies are captured when files are saved, and sent automatically or manually. The last allows members to control when their changes are transmitted as, for example, when they are stable enough to be shared. Note that, whether automatic or manual, transmission only occurs if the working copy has no compilation errors, so the merged system will not be affected by syntactic errors due to invalid files. Additionally, members leaving the team cause their changes to be removed from the merged system.

Changes in the head (check-ins) are processed automatically. Unlike before, where the model is able to ensure that changes are only transmitted if files are syntactically valid, here the team should follow the recommended practice to forbid compilation errors from entering the development line [1], which can be easily supported by modern IDEs and VCSes (e.g., via pre-commit hooks).

This model is very flexible and supports mainstream VCSes and their workflows found today in practice.

##### B. Merged System

Albeit physically made of folders and files, the system under construction and the merged system are abstractly modeled as a *tree of typed and attributed nodes*, like in Fig. 3 for the part that constitutes the file `Primate.java`.

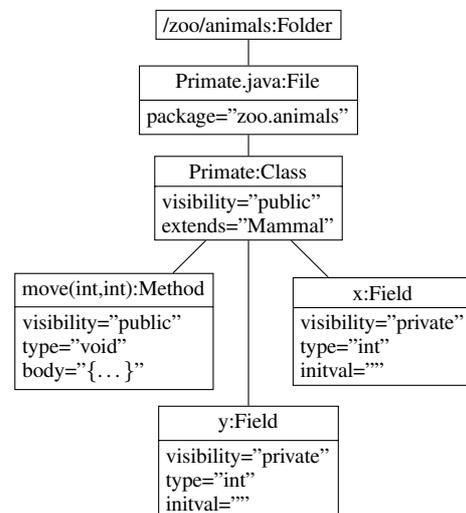


Figure 3. The abstract model of the system.

Every folder, file, and program element inside a file is a node having a type and a set of attributes (none for folders) in the system’s tree. The possible types and attributes are specified by the domain, which in the figure is that of Java.

This system model allows us to compare two states of a file to determine differences at node and attribute level. This is important to track the changes made to the nodes and

their attributes by every member and at the head, which are used for background merging and to find members to report conflicts. Members’ changes are computed by comparing working copy files with their bases in the development line while head’s changes are computed by comparing successive file states at the head. In addition, the model is extended with edges (not shown for clearness) representing dependencies between program elements, like the “extends” dependency from “Primate” to “Mammal”. Thus, it forms an abstract semantic graph used for analysis of the merged system (explained later).

The merged system is updated in the background by merging the most recent changes using structural merging [17]. This collects the changes of each node in the system’s tree, merges them, and returns the new merged state, which is used to rewrite the node in the merged system. This is done for every recently changed node in the system’s tree, thus resulting in an updated merged system.

Structural merging allows fine-grained merges of source files to be done automatically in places where a textual merge would require manual intervention. For example, changes to different program elements or attributes of a program element in the same line of a source file are merged automatically, contrasting with a conflict on the line given by textual merging. For example, Fig. 4 shows the changes made by Anne and Bob in their working copies of a base file F.java (the rectangles and strikes), and the resulting file in the merged system. Note that structural merging was able to merge the changes of the attributes of “class F” and “float pi”, despite being on the same line. Likewise, additions of “int a” and “int b” at the same line are handled transparently because structural merging recognizes them as different program elements. The gray color represents temporary resolutions of structural conflicts done in the merged system, as explained next.

### C. Detection of Conflicts

Our solution detects conflicts of different types, namely, structural, language, behavior, and test conflicts.

1) *Structural Conflicts*: These conflicts are detected during background merging, and they are temporarily resolved in the merged system using *default resolutions* to not halt background merging — this does not change the code in the working copies.

A *pseudo direct conflict* occurs when different attributes of a node are concurrently changed, or the same attribute is simultaneously changed to the same value. In fact, it is a warning that reminds there is no structural conflict, but only the possibility of a semantic conflict — probably, it may be detected as a language conflict (see below). In Fig. 4, this conflict occurs in “class F” and “float pi” after Anne and Bob changed these program elements’ attributes.

The *attribute change & change conflict* occurs when the same attribute of a node is concurrently changed to different

<p>F.java (base)</p> <pre>class F { public float pi; int q = 1; int m() { return 1; } }</pre>	<p>F.java (Anne)</p> <pre>final class F { public float pi = 3.14; int a = 1; int q = 2; int m() { return q; } }</pre>
<p>F.java (Bob)</p> <pre>public class F { public float pi; int b = 2; int q = 3; int m() { return 1; } }</pre>	<p>F.java (merged system)</p> <pre>public final class F { float pi = 3.14; int a = 1; int b = 2; int q = 0; int m() { return q; } }</pre>

Figure 4. File merging example.

values. One example is the “initval” attribute of “int q”, which was changed by Anne to 2 and by Bob to 3. In this case, the default resolution is to assign a *default value* to the attribute in the merged system. Default values are predefined for each attribute type, and for “int” it is zero (in gray).

The *node change & deletion conflict* occurs when there are simultaneous changes and deletions to the same node. This happened with method “m()” changed by Anne and deleted by Bob. In this case, the default resolution is to ignore deletions to preserve changes, so Anne’s change prevails in the merged system. (We tested this decision with twenty-one graduate students by exposing them to a “change & deletion” situation, and they all decided to preserve the change.)

The *inconsistent node type conflict* happens when trying to put nodes of a different type at the same location in the system’s tree. For example, a developer adds a file named “F.java” while another adds a folder named “F.java”. If this occurs, the default resolution is to remove the node from the merged system. In practice this bizarre case should never occur, but the model supports it.

An important advantage of structural merging is that a direct conflict never occurs when different program elements in a file are changed, thus reducing overload by avoiding notifications of false direct conflicts.

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: this is why default resolutions in the merged system are always temporary.

2) *Language Conflicts*: The merged system is automatically compiled every time it is updated. We call the resulting compilation errors *language conflicts* because they are caused by invalid combinations of developers’ changes with

respect to the static semantics of the programming language. One example is the *invisible method conflict* that occurs when a developer makes a method private while another one adds a call to it outside that method’s class. A more complex example is the *undefined constructor conflict* that occurs when a developer adds a constructor with one argument to a class having no constructors as another developer creates a subclass of that class.

By leveraging the compiler to detect language conflicts we avoid re-implementing complex checks of programming language rules: we just listen to the compilation output, process the errors, and report corresponding language conflicts on the user interface (see Section V).

3) *Behavior Conflicts*: These conflicts represent potentially unwanted behavior due to unexpected interactions between concurrent changes. They are detected after updating the merged system by searching for *conflict patterns*, that is, logical conjunctions of facts regarding the program elements and their dependencies in the abstract semantic graph of the merged system, which identify unwanted behavior.

The simplest is the *dependency-based conflict* previously discussed in Section III. Its pattern is

$$\exists x, y \in G : dep^*(x, y)$$

where  $x$  and  $y$  are nodes changed in the graph  $G$  and have a transitive dependency  $dep$ . More specialized patterns are generally more interesting because they represent conflicts that are more difficult to find by developers, like the *unexpected override conflict* in Section II, which can be 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  $B$ .

Note that these patterns are deemed as behavior conflicts only if the facts correspond to nodes changed by different team members (we omitted this part in the above definitions to avoid complicating them).

An advantage of this model is that conflict patterns can be easily added to support an increasing number of complex behavior conflicts (e.g., “unexpected dynamic binding”), and are reusable across projects. Moreover, conflict patterns contrast with tests, which have to be designed and only detect conflicts on the features they test.

4) *Test Conflicts*: A test conflict is a test that fails after updating the merged system and its execution flow has reached two or more methods changed by different members.

Taking the Zoo application, imagine that Anne adds a test to verify if all species have a price returned by method “getPrice()” while Bob creates class Chimpanzee without such method (because he is not aware of Anne’s requirement). Afterwards, the merged system is updated with both changes, and Anne’s test fails when verifying the price of Chimpanzee, as the following execution flow shows:

zoo.testing.ZooTests.setUp()	✓	
zoo.testing.ZooTests.testAnimalGetPrice()	✓	(Anne)
...		
zoo.animals.Animal.getPrice(Ljava/lang/Class;)	✓	
zoo.animals.Chimpanzee.getPrice()	✗	(Bob)

This results in a test conflict for “testAnimalGetPrice()” because its execution flow has a method changed by Anne (the test itself) and a method missing in the class created by Bob. In this example, the test tried to call “getPrice()” via reflection on Chimpanzee. Our solution intercepts reflection calls too and checks if missing methods were deleted or never existed, which was the case for Bob. Consequently, this test conflict is named *missing method conflict*. Tests are very useful to detect conflicts involving reflection, which are hard to find using conflict patterns.

#### D. Reporting Conflicts

Each conflict is reported to the members that changed the node or nodes affected by it. For structural conflicts, the affected node is that at the location of the conflict in the system’s tree; for language conflicts, the affected nodes are the ones involved in the compilation error; for behavior conflicts, they are those that match the corresponding conflict pattern; and for test conflicts they are the nodes which represent the methods in the failed execution flow. Only the members that changed the affected nodes and their attributes are notified of the conflict. Such members are found by looking for who changed the affected nodes in the node change tracking information.

### V. IMPLEMENTATION IN ECLIPSE

Our tool, called WeCode, implements our solution for Java programming inside the Eclipse IDE. WeCode has both client and server plugins, and at the moment it supports Subversion (<http://subversion.apache.org>).<sup>1</sup>

#### A. The Server

This plugin runs on Equinox (<http://eclipse.org/equinox>) and is responsible for managing teams: it handles members’ joins and leaves; tracks their changes to the system’s tree; does background merges; and updates the conflicts affecting the team.

Each team holds a project inside Equinox containing the code of the team’s merged system. This is updated every time changes are received from client plugins or from check-ins in the team’s development line, which is monitored by the server.

The merged system is automatically compiled by the Eclipse Java Tools installed in Equinox. The server listens for marker deltas (`IMarkerDelta`) corresponding to

<sup>1</sup>Subversion was our first choice because it had the best plugin for Eclipse when we started development. Currently, we plan to support other VCSes, like Git and Mercurial (<http://mercurial.selenic.com>).

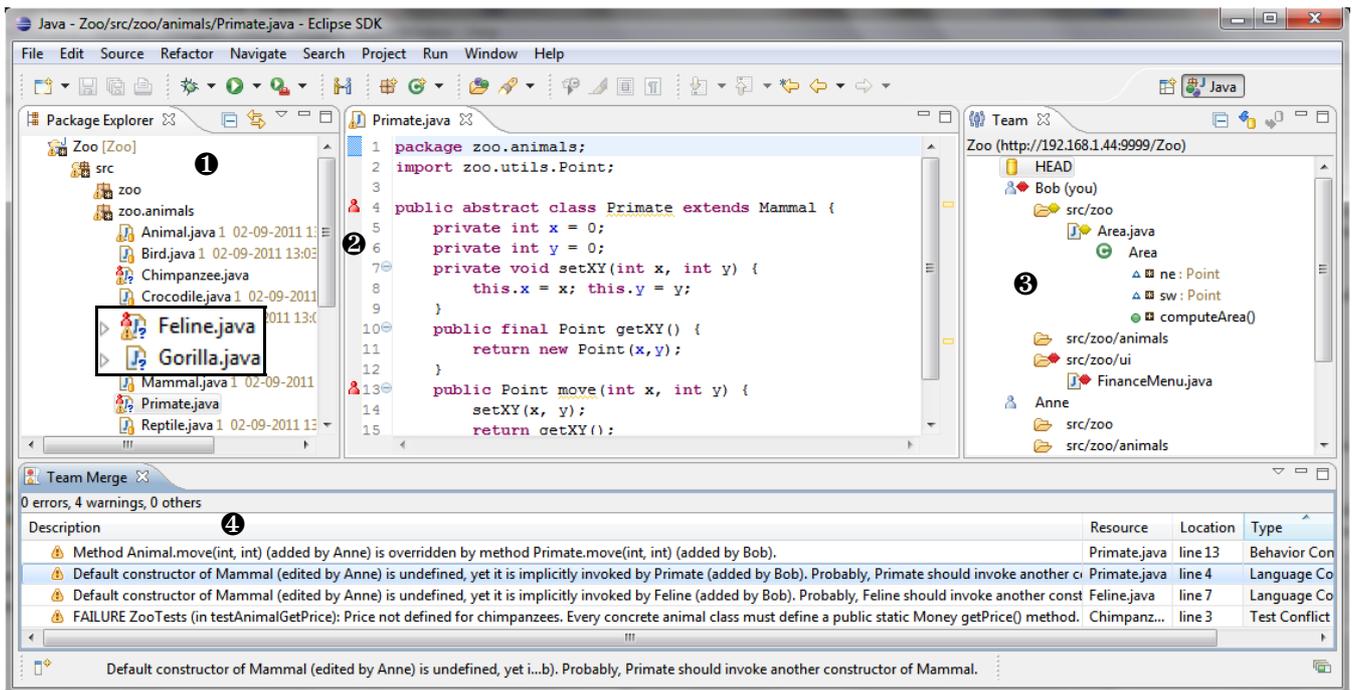


Figure 5. Continuous merging inside the IDE.

errors in the compilation output to update the language conflicts affecting the team.

At the same time, the server compares the previous and the updated states of the merged system to determine how it was modified during background merging. This is done to find out which conflict patterns need to be verified to update the behavior conflicts affecting the team. In what concerns the detection of behavior conflicts, we leverage the abstract semantic graph implemented by the Eclipse Java Tools to look for instances of conflict patterns in the merged system.

Test conflicts are found by running JUnit test cases (<http://www.junit.org>) after updating the merged system. To support this, we created modified versions of the `org.eclipse.junit.{core, runtime}` plugins to collect the execution flows of running test cases. In particular, we have a version of class `RemoteTestRunnerClient` (core plugin) installed on the server JVM, and a version of class `RemoteTestRunner` (runtime plugin) installed on the test JVM, which opens a socket connection to the server and sends back the results as tests are run.

The server launches the test JVM with an agent library that instruments the methods' entry points in all loaded classes (except those of the Java runtime, JUnit, and Equinox/Eclipse) with code that logs method calls. Test execution flows are then collected by logging the method calls between each test start and end.

Every conflict detected by the server contains a detailed message that describes what happened, the affected program

elements, and the affected members that have modified those elements, thus speeding up conflict resolution on the client side. All conflict updates are sent to the client plugin of affected members.

### B. The Client

This plugin collects changes to folders and files in the developer's working copy by listening to events sent by the Subversion plugin and the Eclipse IDE, and sends them to the server. In addition, it receives from the server the latest changes of other members and the most recent conflict updates. All these sends and receives are automatic or by developer's request. The client plugin also offers the following two views that constitute the main of WeCode's user interface.

1) *The Team View*: Following Fig. 5, this view (③) lists all members in the team (including the head), and details their changes to folders, files, and program elements down to fields and methods. This view has buttons that developers may use to publish their changes (↕) and receive updates (↕) within their team (located in the server at the URL).

The Team view uses red (◆) and yellow (◆) icons to mark files simultaneously changed by the developer and other members. The red icon alerts that there are structural conflicts (other than pseudo) inside the file while the yellow icon indicates that even though they modified the same file, none or only pseudo direct conflicts exist. This color scheme avoids developers wasting time investigating notifications by helping them focus on "urgent" files. In addition, the icons

are kept minimal to avoid overloading the user interface: we only show them in the developer’s subtree and at folder and file nodes. (Supporting icons for other conflict types or at finer-grained nodes might be distracting in this view but we need to investigate it further.)

Developers can use this view to compare and exchange code among them or between them and the head. For example, they can access the file compare editor of any file node to integrate other members’ changes, so that they resolve conflicts while changes are fresh in their minds or stay updated with the most recent code within the team. They can also use a chat view<sup>2</sup> to ask other members if their changes are already stable, thus avoiding merging unfinished code. For instance, a true direct conflict (red icon) can be collaboratively resolved by merging other members’ changes, publishing the locally merged file, and asking those members to accept the merged file, thus turning the icon yellow.

2) *The Team Merge View*: This view (④) lists all language, behavior, and test conflicts affecting the developer, which in the figure are those already introduced in the paper.

Every conflict has a detailed description that reports its nature, the affected program elements, the affected members, and how these changed the affected elements. The description results from instantiating the placeholders of the conflict’s template. For example, the template for tests conflicts includes the message returned by the failure, which for the test case in the figure is the message in `assertNotNull("Price not ...");`.

Developers can double click a conflict in the Team Merge view to quickly jump to its location, which can be a program element or a statement inside a method. In addition, conflicts are signaled in the Package Explorer (①), and inside editors at their location in the affected files (②) (for an example of a conflict at statement level see line 6 of Fig. 7 in [12]).

All the information is unobtrusively presented to developers in a way that resembles the continuous compilation experience in Eclipse’s Problems view. With these features, the Team Merge view helps developers speed up resolution, instead of wasting time investigating many change notifications to detect conflicts, as discussed back in the Problem section (Section III).

### C. Preparing for the Evaluation

To support the evaluation of our tool, we created an additional view, called Team Alerts (Fig. 6), which uses dependency analysis to offer a heuristic mode of conflict detection similar to those in [13], [14], [15]. This view notifies the developer about co-workers’ changes to the APIs referenced by files modified by the developer. The notifications are listed for the file selected in the Package Explorer or opened in the active editor.

<sup>2</sup>The Collaboration view in Eclipse ECF (<http://www.eclipse.org/ecf>).

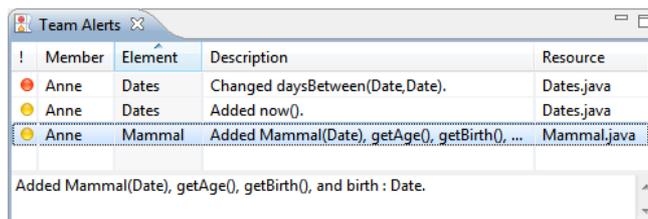


Figure 6. Team Alerts view showing notifications for Feline.java.

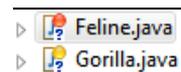


Figure 7. Notifications are signaled in the Package Explorer view.

For example, Fig. 6 lists the notifications for Feline.java. The red icons alert that a co-worker changed program elements used in Feline.java whereas the yellow icons alert for changes to unused program elements in types referenced by Feline.java. The notifications may be double clicked to open an editor to compare the co-worker’s file (that in the “Resource” column) with its corresponding local file. The icons are also shown for files in the Package Explorer (red icons win) to alert developers when they are focused somewhere else, as shown in Fig. 7.

## VI. EVALUATION

The evaluation here described shows that our solution does not overload developers with notifications and improves early detection of conflicts when compared to existing approaches based on change and dependency-based awareness.

### A. Experimental Design

To show this, we ran several controlled user experiments comprising three treatments that correspond to different levels of support for conflict detection: the REPOSITORY had no support, the HEURISTIC was supported by the Team Alerts view, and the MERGE used the Team Merge view. Using this scheme we wanted to assess the number of conflicts detected in each treatment before check-in, and the notification overload (this one for the last two treatments).

The experiments involved twenty-one graduates in computer engineering from our university, one half being recent bachelors and the other being PhD students, having enough experience with the tools used in the experiment, namely, Eclipse, Java, and Subversion.

The experiment was ran once for each team comprising one subject and one confederate (the experiment host). The teams were randomly assigned into the treatments so that the number of bachelors and PhDs was the same for all treatments.

Subjects were told they were going to be studied on how they managed conflicts between concurrent programming tasks modifying the Zoo application (a total of 41 classes

and 1143 LOCs), which was sent to them at least two days before their session. At the beginning of their experiment, they watched a treatment-specific video showing the tools using a conflict that would not occur during their session. Then, they were given a guide with six tasks and the code to type in each task. They were told to follow the task order and say aloud when a task started or finished and when a conflict was detected. All sessions were recorded with prior subject agreement. At the end, a questionnaire containing 10-point Likert scale and free response questions was given. All this was designed so that each experiment took less than one hour and thirty minutes.

The confederate followed a list of concurrent tasks, which inserted the following indirect conflicts at about the same time for all subjects: two undefined constructor conflicts (language), one unexpected override conflict (behavior), and one missing method conflict (test). All conflicts were inserted before subjects had finished half of their work so that they had enough time to detect them before the end. The confederate also checked in after every task in the REPOSITORY case; this was unnecessary in the other cases because the tool was configured to automatically publish changes to the team’s server.

## B. Results

1) *Quantitative Analysis:* Table I shows the number of conflicts detected (D) and not detected (ND) in each treatment before subjects checked in at the end of their tasks — a conflict was considered detected when (MERGE) subjects said they had seen it in the Team Merge view, and when (HEUR. & REPO.) subjects said something like “I think there is a problem: I changed this and my co-worker changed that” and the “problem” was a conflict.

*What is the effect of the awareness mode on the ability of developers to detect conflicts early?* Table I shows that REPO. subjects found none of the conflicts before check-in. Only one subject synchronized frequently with the repository, but he only looked for direct conflicts at file level. In general, all REPO. subjects were observed during check-in to pay attention only to those files that they and the confederate also changed, thus missing the indirect conflicts. The HEUR. subjects did better, but most only detected conflicts after importing changes from their co-worker, mainly “to stay updated with the most recent code” as one said, thus producing a compilation error in their working copies that caught their attention. Only one succeeded in finding the unexpected override conflict by correlating the additions of two methods in the hierarchy. In contrast, MERGE subjects were able to detect all conflicts early on given the detection capability provided by this mode. In general, they started resolution at their best opportunity, generally between their tasks, and only then they decided to import from their co-worker as needed to resolve the conflicts.

Table I  
NUMBER OF CONFLICTS DETECTED IN EACH TREATMENT. THE EFFECTS OF THE TREATMENTS ARE STATISTICALLY SIGNIFICANT ACCORDING TO PEARSON  $\chi^2$  TEST ( $p < .05$ ).

	MERGE	HEUR.	REPO.	Pearson $\chi^2$	df	p
D	28	7	0	62.4	2	.001
ND	0	21	28			

The results in Table I provide evidence that repository support and dependency-based awareness are not sufficient to support early detection of indirect conflicts, and that the continuous merging approach has greater potential.

2) *Qualitative Analysis:* Table II shows the scores of the Likert questions in the questionnaire.

Q1’s scores show that subjects had a very different perception about the number of false positives presented in the two modes, which is consistent with the kind of support in these modes.

Q2’s scores express a strong wish of being informed about conflicts during programming, and interestingly the subjects that scored higher in such feature were those who experienced it, thus reinforcing the usefulness of continuous merging. When asked to justify their score, the subjects responded:

“[...] we could avoid spending too much time resolving conflicts at the end [check-in]. Additionally, it would help resolve conflicts while we still remembered the code where they happened.” (HEUR., Q2=9)

“Allows to manage conflicts as they occur, and does not let changes grow.” (MERGE, Q2=9)

“It allowed me to reduce the time to resolve conflicts [...], instead of a slow commit later.” (MERGE, Q2=10)

Regarding this question, one said “I did not give maximum score because I occasionally paused my work to check if the conflict was important to resolve right away” (MERGE, Q2=9), and another said “Early detection is crucial to avoid wasting too much time during resolution. Although it may cause distraction, I believe with practice it is possible to manage distractions” (HEUR., Q2=8). In general, subjects were able to manage interruption by paying more attention to awareness information after file saves and between tasks. Still, we think interruption management requires further research.

Q3’s scores summarize subjects’ overall experience with the awareness modes in our tool, suggesting they really appreciated knowing what was happening around them that would help coordinate with others. The questionnaire ended by asking subjects to express their opinion about the tool and suggest improvements:

“I liked to know in real-time who was changing the code I was using and where” (HEUR.)

“I appreciated being informed about remote changes with different levels of severity” (HEUR.)

Table II  
 QUESTIONNAIRE SCORES IN 10-POINT LIKERT SCALE (“1-STRONGLY DISAGREE”, 10-“STRONGLY AGREE”). THE VALUES ARE SHOWN AS “MEAN (STD. DEV.)”, AND “N.A.” FOR NOT APPLICABLE.

	MERGE	HEUR.	REPO.
Q1: The mode’s view listed many false positives, that is, alerts not corresponding to real conflicts, thus distracting me.	2.1 (0.99)	7.6 (1.36)	n.a.
Q2: I (liked / would like) to be informed about conflicts while I am programming, instead of being warned only later at check-in.	9.3 (0.70)	7.7 (1.70)	8.00 (1.85)
Q3: I would use this mode’s view and recommend it to other colleagues.	9.0 (0.93)	7.8 (0.75)	n.a.

“Some changes that appeared as yellow at the beginning revealed to be problematic [at check-in]” (HEUR.)

“I liked less the fact that the tool did not signal the potential compilation error due to the concurrent addition of constructors” (HEUR.)

“I liked the icon in the editor informing me about the conflict and with whom I was conflicting” (MERGE)

“I liked most its simplicity of use” (MERGE)

“I liked tests being run on the merge of the code of the entire team” (MERGE)

This qualitative part shows that subjects deeply appreciate being informed about conflicts during programming, and that there is a tendency to favor continuous merging.

### C. Threats to Validity

Every experiment is challenged by threats to its construct, internal and external validity.

1) *Construct validity*: We think that the application, tasks, conflicts, and questionnaires used in our study are valid by construction to evaluate our tool and the effect of the different awareness modes on the ability of developers to detect conflicts. Subjects also knew nothing about which conflicts would occur or that our tool was being evaluated so as to avoid influencing them.

2) *Internal validity*: The confederate was used to prevent confounds caused by varying behaviors of genuine co-workers that might have influenced subjects’ behavior. The subjects also had no previous experience with our tool, so there were no learning effects on their performance, and we neutralized programming skills as much as possible by giving the code to type in each task. The best we could do to avoid personal characteristics (e.g., curiosity) from confusing the results was to randomly select subjects into treatments. As such, we think our study has internal validity.

3) *External validity*: The major threat to the generalization of our study’s results is that changing a simple application, like Zoo, by typing pre-written code is not representative of real practice. Regarding this, our study did

not cover aspects due to the complexity of real software development, which may influence how programmers use and benefit from a conflict detection tool. These include project aspects like size and geographical distribution, and tool aspects such as usability and interruption management, so studies in real projects are needed. Our study might be threatened because of our choice of conflicts. The results show that our solution performs better than others for the chosen conflicts. However, we think that it will not perform worse for other conflicts, especially due to its detailed detection capability. Even though we need to understand the kind of merge conflicts occurring in real projects, it is reasonable to assume that indirect conflicts will be at least as complex and hard to detect as those in the study, so we think our automatic conflict detection solution will be advantageous in practice. In addition, our choice of tasks might be threatened in terms of the false positives they did or did not generate despite our efforts to avoid bias in any way. The use of students seems reasonable considering one study reporting students and professionals have no major difference in understanding dependencies and relationships in software [18], which is important to find conflicts between concurrent changes. To sum up, our results are indicative of the benefits of our solution and suggest that a longitudinal study in the industry is necessary to ground a theory of awareness of software conflicts.

## VII. RELATED WORK

This section outlines several tools related to our work, which provide awareness of software changes.

Tools that provide awareness of direct conflicts at file granularity may overload the developer because of changes to independent program elements inside the same files [15], [16], [19], [20]. A fine-grained solution like ours does not have this shortcoming.

Other tools go beyond these and support dependency-based awareness. Tukan [13] signals developers with the presence of co-workers and the changes they made in elements near a dependency path from the element focused on by the developer. Presence signals help find co-workers for collaboration, and change signals help prevent direct and indirect conflicts. Signals are ranked according to a dynamic function of path length and relevance, which by default emphasizes more the shorter paths as these are assumed to connect elements having a stronger dependency. This tool does not have a place where all signals are listed, so developers are expected to contact co-workers or to inspect concurrent changes once they see a signal of interest, before making changes to the focused element; otherwise, it can be difficult to remember where signals exist in the code for later investigation.

CollabVS [14] provides presence awareness and notifies about direct and indirect conflicts involving elements connected by an unlimited dependency path. There is no ranking

mechanism to help developers focus on the most interesting notifications. Developers can select the element granularity at which conflicts are detected to that of file, type, or method, but in practice it can be difficult to determine which works best at each moment [21]. Developers upon receiving a notification may set a “watch” on the element edited by the co-worker to remind the developer to check for conflicts after some time or after the co-worker removes focus from the element. However, this mechanism may provoke disturbance since “watches” need to be requested to co-workers, timeouts can be hard to set properly, and co-workers may enter and leave elements frequently.

Palantír [15] notifies an indirect conflict when a file modified by the developer depends on a file that had its public APIs altered by a co-worker. Its main focus is on syntactic indirect conflicts. Notifications carry the modifications to the public APIs of types in remotely changed files. In addition, a notification is a “bomb” if there is a dependency on a code element (type or method) that had its public signature edited in the remote file, otherwise, they are just a “warning” of possible interesting changes to the remote file’s public APIs. This inspired our design of WeCode’s Team Alerts view. Notifications are also “red” or “yellow” respectively when the remote file was checked in or is still being changed by a co-worker. This tool does not support indirect conflicts involving remote files not present in the developer’s working copy — these are files created by co-workers or that the developer did not check out.

Both CollabVS and Palantír do not notify conflicts involving check-ins bypassing the tool or done before developers checked out via the tool. This is supported by WeCode’s model of team work in a development line.

All the above tools only support direct conflicts and dependency-based indirect conflicts. In addition, developers must investigate notifications to determine if conflicts really exist. In contrast, WeCode considers the unpredictable semantics that result from merging the changes to an object-oriented system made by a whole team. Besides using analysis, WeCode uses a merged system to detect complex conflicts via compilation and testing, which is a feature that awareness tools like the above do not provide. For example, testing can detect conflicts which are very hard or undecidable via analysis, like involving code reflection.

YooHoo [22] reports API changes that may break the code or that are of interest to the developer as determined by the dependencies within the files owned (recently committed) or selected by the developer. This tool is not designed to detect conflicts, but to help developers adapt their code to the evolution of APIs in external projects or in branches of sub-teams within a large team.

Crystal [4] does separate background merges of pairs of repositories, comprising that of the developer and that of a co-worker or a central master. The result is one of “textual merge failure”, “build failure”, “tests failure”, or “tests

passed” relationship for each repository pair. This tells each pair of developers if their mutual merge is problematic, but does not tell them which conflicts are exactly occurring, so they have to interrupt their work and spend time discovering conflicts. In contrast, WeCode does a single background merge of all developments of a team working on the same branch and informs inside the IDE about the precise details of the conflicts affecting the team as a whole. This allows us to catch complex conflicts involving two or more developers, and avoids duplication of awareness icons caused by the same conflict in multiple merged repository pairs. WeCode does not distract developers because of textual merge failures that are easily handled via structural merging. Our solution also supports uncommitted changes, so conflicts can be found as soon as developers want to be informed.

Three studies compared support for early conflict detection against not having such support [14], [16], [23]. Like them, we concluded that users like to have support for early conflict detection, but unlike them our study also compared two support levels.

## VIII. CONCLUSION AND FUTURE WORK

The problem of merge conflicts in collaborative programming is an important one as they are known to cause software defects. The industry and research recognize this problem and that a conflict detected earlier is much easier to resolve than when detected later at check-in or production.

Awareness has been proposed to help developers detect conflicts earlier. However, all known approaches require developers to detect conflicts by themselves and may overload them with notifications, hence making detection difficult.

We have proposed a novel solution that precisely and accurately detects conflicts on behalf of developers, thus avoiding overloading them with notifications. It introduces the notion of continuous merging inside the IDE, which enables earlier resolution of conflicts while developers still have their changes fresh in their minds, making resolution easier. An empirical evaluation confirmed that our solution improves early detection of conflicts and avoids overloading developers in comparison with existing approaches.

There are several directions for our future work. First, we will continue improving WeCode’s collaborative features and usability. Second, we want to do a longitudinal study with professional programmers in an industrial setting to determine the influence of continuous merging in their software process, and to check if new collaborative patterns emerge. Third, we want to measure the overall effect of continuous merging on software quality.

## ACKNOWLEDGMENT

The first author was supported by FCT scholarship SFRH/BD/27652/2006. This work was also supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds.

## REFERENCES

- [1] S. Berczuk and B. Appleton, *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Addison-Wesley, 2002.
- [2] M. Fowler. Continuous Integration. [Online]. Available: <http://martinfowler.com/articles/continuousIntegration.html>
- [3] D. E. Perry, H. P. Siy *et al.*, “Parallel changes in large-scale software development: An observational case study,” *ACM Trans. Softw. Eng. Methodol.*, vol. 10, pp. 308–337, July 2001.
- [4] Y. Brun, R. Holmes *et al.*, “Proactive Detection of Collaboration Conflicts,” in *ESEC/FSE '11: Joint Meet. of the Euro. Softw. Eng. Conf. and the Inter. Symp. on the Foundations of Softw. Eng.* ACM, 2011, pp. 168–178.
- [5] P. Duvall, S. M. Matyas *et al.*, *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, 2007.
- [6] R. Grinter, “Using a Configuration Management Tool to Coordinate Software Development,” in *COOCS '95: Conf. on Organizational Computing Systems*. ACM, 1995, pp. 168–177.
- [7] C. de Souza, D. Redmiles *et al.*, “”Breaking the code”, Moving between Private and Public Work in Collaborative Software Development,” in *GROUP '03: Inter. Conf. on Supporting Group Work*. ACM, 2003, pp. 105–114.
- [8] P. Dourish and V. Bellotti, “Awareness and Coordination in Shared Workspaces,” in *CSCW '92: Conf. on Computer Supported Cooperative Work*. ACM, 1992, pp. 107–114.
- [9] D. Damian, L. Izquierdo *et al.*, “Awareness in the Wild: Why Communication Breakdowns Occur,” in *ICGSE '07: Inter. Conf. on Global Softw. Eng.* IEEE Computer Society, 2007, pp. 81–90.
- [10] S. R. Fussell, R. E. Kraut *et al.*, “Coordination, Overload and Team Performance: Effects of Team Communication Strategies,” in *CSCW '98: Conf. on Computer Supported Cooperative Work*. ACM, 1998, pp. 275–284.
- [11] M. Kim, “An Exploratory Study of Awareness Interests about Software Modifications,” in *CHASE '11: Workshop on Cooperative and Human Aspects of Softw. Eng.* ACM, 2011, pp. 80–83.
- [12] M. L. Guimarães and A. Rito-Silva, “Towards Real-Time Integration,” in *CHASE '10: Workshop on Cooperative and Human Aspects of Softw. Eng.* ACM, 2010, pp. 56–63.
- [13] T. Schümmer and J. Haake, “Supporting Distributed Software Development by Modes of Collaboration,” in *ECSW '01: Euro. Conf. on Computer Supported Cooperative Work*. Kluwer Academic Publishers, 2001, pp. 79–98.
- [14] P. Dewan and R. Hegde, “Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development,” in *ECSCW '07: Euro. Conf. on Computer Supported Cooperative Work*. Springer, 2007, pp. 159–178.
- [15] A. Sarma, G. Bortis *et al.*, “Towards Supporting Awareness of Indirect Conflicts Across Software Configuration Management Workspaces,” in *ASE '07: Inter. Conf. on Automated Softw. Eng.* ACM, 2007, pp. 94–103.
- [16] J. T. Biehl, M. Czerwinski *et al.*, “FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams,” in *CHI '07: Conf. on Human Factors in Computing Systems*. ACM, 2007, pp. 1313–1322.
- [17] J. P. Munson and P. Dewan, “A Flexible Object Merging Framework,” in *CSCW '94: Conf. on Computer Supported Cooperative Work*. ACM, 1994, pp. 231–242.
- [18] M. Höst, B. Regnell *et al.*, “Using Students as Subjects – A Comparative Study of Students and Professionals in Lead-Time Impact Assessment,” *Empirical Softw. Eng.*, vol. 5, pp. 201–214, November 2000.
- [19] G. Fitzpatrick, P. Marshall *et al.*, “CVS Integration with Notification and Chat: Lightweight Software Team Collaboration,” in *CSCW '06: Conf. on Computer Supported Cooperative Work*. ACM, 2006, pp. 49–58.
- [20] S. Hupfer, L.-T. Cheng *et al.*, “Introducing Collaboration into an Application Development Environment,” in *CSCW '04: Conf. on Computer Supported Cooperative Work*. ACM, 2004, pp. 21–24.
- [21] P. Dewan, “Dimensions of Tools for Detecting Software Conflicts,” in *RSSE '08: Inter. Workshop on Recommendation Systems for Softw. Eng.* ACM, 2008, pp. 21–25.
- [22] R. Holmes and R. J. Walker, “Customized Awareness: Recommending Relevant External Change Events,” in *ICSE '10: Inter. Conf. on Softw. Eng.* ACM, 2010, pp. 465–474.
- [23] A. Sarma, D. Redmiles *et al.*, “Empirical Evidence of the Benefits of Workspace Awareness in Software Configuration Management,” in *SIGSOFT '08/FSE-16: Inter. Symp. on Foundations of Softw. Eng.* ACM, 2008, pp. 113–123.