Collaborative Software Development on the Web

Martin Nordio, H.-Christian Estler, Carlo A. Furia, and Bertrand Meyer

ETH Zurich, Switzerland `firstname.lastname@inf.ethz.ch`

Abstract. Software development environments (IDEs) have not followed the IT industry's inexorable trend towards distribution. They do too little to address the problems raised by today's increasingly distributed projects; neither do they facilitate collaborative and interactive development practices. A consequence is the continued reliance of today's IDEs on paradigms such as traditional configuration management, which were developed for earlier modes of operation and hamper collaborative projects. This contribution describes a new paradigm: cloud-based development, which caters to the specific needs of distributed and collaborative projects. The CloudStudio IDE embodies this paradigm by enabling developers to work on a shared project repository. Configuration management becomes unobtrusive; it replaces the explicit update-modify-commit cycle by interactive editing and real-time conflict tracking and management. A case study involving three teams of pairs demonstrates the usability of CloudStudio and its advantages for collaborative software development over traditional configuration management practices.

1 Introduction

The Integrated Development Environment is the software developer's central tool. IDEs have undergone considerable advances; their fundamental structure and mode of operation are still, however, what they were decades ago. In particular, while Internet development has benefitted from IDEs, the IDE has not benefitted from the Internet; it remains an essentially personal tool, requiring every member of a project to work on a different copy of the software under development and periodically to undergo a painful process of reconciliation.

CloudStudio, the IDE described in this article, brings software development to the Internet. In recent years ever more human activities, from banking to text processing, have been "moved to the cloud". CloudStudio does the same for software engineering by introducing a new paradigm of software development, where all the products of a software project are shared in a common web-based repository.

Moving software development to the cloud is not just a matter of following general trends, but a response to critical software engineering needs, which current technology does not meet: supporting today's distributed developments, which often involve teams spread over many locations, and iterative development practices such as pair programming and online code reviews; maintaining compatibility between software elements developed by different team members;

avoiding potentially catastrophic version incompatibility problems; drastically simplifying configuration management.

CloudStudio brings flexibility to several new facets of software development, most importantly configuration management (CM): to replace the traditional and painful update-modify-commit-reconcile cycle, CloudStudio tracks changes at every location in real time and displays only the selected users' changes in the integrated editor. The compiler and other tools are aware of the current user preferences, and target the version of the code coinciding with the current view. CloudStudio also integrates communication tools (a chat box and Skype), and includes a fully automated verification component, including both static (proof) and dynamic (testing) tools. This array of tightly integrated tools makes CloudStudio an innovative IDE, which can improve the quality and speed of projects involving distributed teams, and support highly collaborative development practices.

CloudStudio is an ambitious project for which we have built a prototype, which readers can try out (see Section 5). To demonstrate CloudStudio's potential to facilitate distributed development, we have conducted a case study where three teams of two programmers modified and extended existing software projects with CloudStudio and with traditional CM (i.e., Subversion). Within the limits given by its limited extent, the case study substantiates the claims that CloudStudio can facilitate collaborative development without interfering with the standard habits of programmers. While the initial results from the prototype are exciting, many research challenges remain. This article describes both the current CloudStudio framework and the open research challenges that lie ahead.

Section 2 presents the challenges of collaborative and distributed development. Section 3 is an overview of CloudStudio from the user perspective. Section 4 describes CloudStudio's CM model and awareness system. Section 5 briefly describes the current implementation and the verification features. Section 6 presents a case study used to evaluate CloudStudio's potential for collaborative development. Section 7 discusses related work; Section 8 summarizes and discusses future work.

2 Distributed and Collaborative Development

Today's software projects are increasingly multipolar. "Gone are the days of one-company, one-site projects; most industry developments involve teams split over several locations, countries, cultures" [15]. Such projects involve not just developers but many other stakeholders with different backgrounds and needs, from users and managers to testers and trainers. Organizationally, they no longer limit themselves to a single location or even a single company but follow talent wherever it is, increasingly leading to a distributed mode of development.

Such distributed projects raise a full set of new software engineering challenges [11,10,9], which the standard approaches do not address well. Examples of these challenges include requirements and interface specification in the context of distributed development. Many failures have been reported in outsourced and distributed projects, often due not to lack of technical expertise, but to difficulties in management and communication. Distributed projects require new methods and sophisticated tool support to handle the complex interactions between the many actors involved.

An orthogonal trend that brings its own challenges is the growth of methods based on iterative, incremental, and highly collaborative development, such as agile methods. These approaches advertise informal collaboration and continuous direct communication between team members as solutions to the deficiencies of traditional structured development processes. Whether and how intense collaboration is achievable when programmers do not sit in the same room are, however, open question; and even for developers working at the same location, tools specifically designed to facilitate collaborative development are still largely unseen.

A central issue, often playing a major part in project failures, is configuration management. CM addresses fundamental needs: making sure that all project members use the same reference versions of every software element, avoiding conflicts as they change various parts of the system, avoiding configuration errors (where version n of module A uses the wrong version of module B), avoiding regression errors (where a previously corrected bug reappears as a result of bad information flow), allowing the re-creation of a previous version of the system or one of its modules.

The initial impetus for CloudStudio was our experience with distributed software development both in the context of a long-running industry development, distributed over many sites and led by the last author, and with a university course which we have taught for several years with a distributed collaborative project involving student teams from several universities [16]. We found that today's tools are badly lacking in support for such distributed setups:

- Communication is a critical issue. Tools such as Skype, WebEx, GoogleDocs, and wikis are useful but not meant for software development.
- Configuration management, the key day-to-day practical issue, is a major hurdle. While CM is essential in any team effort, the tools, based on 30-year-old concepts, are heavy to use (requiring constant "update" and "commit" operations) and poorly adapted to modern distributed projects. These operations distract developers from their truly important tasks. Between the time a developer checks out a component and checks it back in, the project manager and the rest of the team have no idea of what is happening to it; if two developers modify the same component, conflicts will be detected late and will be hard to reconcile. There is always a tendency to branch, often leading to a catastrophe down the line at the time of merging. (Unlike physicists, software developers have their Big Bang at the end.)

Our vision is a new paradigm for software development, both addressing the needs of distributed projects and taking advantage of distribution. The vision is embodied in an experimental distributed software development environment, CloudStudio, allowing teams to work on a common product regardless of their geographical location. Instead of running on each developer's machine, CloudStudio is hosted on the Web and works on a *shared project repository*. The result is a radically new approach providing developers and managers, at any time, with an accurate and up-to-date picture of the entire project. It also includes a profound rethinking of the fundamental task of configuration management, which becomes an unobtrusive automatic technique for keeping track of changes on the developers' behalf and reconstructing earlier versions on demand.

Characteristics of CloudStudio include:

– Unobtrusive configuration management: CloudStudio gives each developer the appearance of having a private copy of the project, but the project is "in the cloud", its material shared between all project members. There is practically no need for traditional update and commit: CM happens in the background as a result of editing actions.
– Awareness system: CloudStudio keeps track of all the changes introduced by the developers, and lets any developer display the changes of the other developers. CloudStudio allows for compiling and verifying the project including/excluding these changes. Thus, a developer's modifications do not block others.
– CloudStudio includes verification tools to improve software quality, based on our related work on "verification as a matter of course" [19].

3 A Session with CloudStudio

This section gives an overview of CloudStudio from the perspective of two users—Stu and Claudia—who are working on the same project from different locations.

After logging in on `www.cloudstudio.ethz.ch/demo` and selecting a project, Stu reaches the main CloudStudio window, pictured in Figure 1. The central frame shows the source code for the current class (*PARAGRAPH*), which Stu can change with the class browser in the right-hand vertical frame. The bottom frame displays the results of the latest compilation and verification runs.

Stu is editing class *PARAGRAPH* concurrently with Claudia, who is working at a different location. At any time, Stu can show or hide Claudia's changes to the code by toggling a button. When changes are shown, as they are in Figure 1, vertical bars of different colors mark each line of code according to its edit status: orange for lines changed or added by Stu (the current user); blue for lines changed or added by Claudia; red for lines with conflicts, that is edited differently by Claudia and Stu; lines without a colored bar are unchanged by anyone. When he compiles the project, Stu can target the base version of the code (only unchanged lines), or include his or Claudia's changes to it, or both. This mechanisms make Claudia and Stu aware of each other's work; they do not have to block and immediately resolve conflicts, but they can continue working without stomping on each other's feet.

CloudStudio offers tools not only to detect and prevent conflicts, but also to resolve them. Stu can see that Claudia is online in the left-hand top frame; he

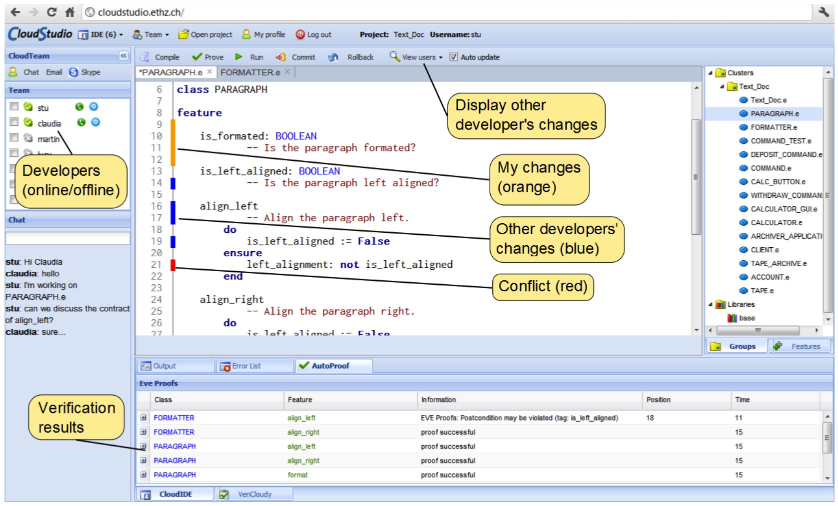

Fig. 1. Main CloudStudio window for user Claudia (balloons are not part of the user interface but of this figure's caption).

can call her on Skype, or chat with her directly in the left-hand bottom frame. After agreeing on what to do with the conflicts, Stu clicks the *commit* button to force a synchronization with Claudia. CloudStudio's explicit commit works quite differently than in standard IDEs: the advanced features for configuration management make its usage quite infrequent. When Stu commits, CloudStudio synchronizes the base version to Stu's current version; if lines with conflicts remain, the commit conservatively skips them so that the base version is in a consistent state. The IDE shows the current base version number n—corresponding to "unchanged" lines of code in the editor—in the top-left corner (**IDE** (n)).

Chatting and talking can become more effective if Claudia and Stu have a means to type collaboratively on the very same piece of code, and to see it change to reflect the edits by both. To this end, CloudStudio offers the *interweave* editing mode where participants work on the code as if they were sitting at the same keyboard. With interweave mode on, the notion of conflict disappears, because Claudia and Stu are effectively working in the same editor, similarly as in GoogleDocs. Stu can enable or disable interweave editing at any time. In fact, most of the development is carried out without interweaving, which is appropriate for fine-grained conflict resolution but generates too much jumble if used for most concurrent editing.

On top of the tools for collaborative development, CloudStudio features a standard IDE integrated in your browser; it even offers tools for automated verification. The bottom frame of Figure 1 shows the outcome of a call to the static verification tool AutoProof [14] on the current version of the code; Stu can inspect failed verification attempts and accordingly modify the code to correct errors.

4 Unobtrusive Configuration Management and Awareness System

This section presents the major feature offered by CloudStudio to support collaborative development: a configuration management system that is not centered around the rigid notion of revision, and that facilitates concurrent collaborative work by multiple programmers.

4.1 Overview of the Problem

The goal of configuration management is to track and control the evolution of software artifacts—code, imprimis—during project development. The evolution is three-dimensional, since software evolves in time, across developers, and in different modules.

The standard approach to configuration management—implemented by tools such as CVS and Subversion—uses a client/server architecture, where a central repository stores incremental snapshots of the codebase, and every developer is a client of the repository who maintains a local working copy of the code. Synchronization between working copies and the central repository occurs by explicit client request through *update* and *commit* operations. When a client A commits, the content of the central repository is changed to include A's changes present in its working copy. Conversely, when a client B updates, B's working copy is updated to coincide with the central repository's. Even if so-called distributed version control systems—such as Git—do not use a client/server architecture, they still require manual operations, comparable to updates and commits, to synchronize a local copy with others. This paradigm makes *conflicts* likely to occur whenever two developers work on the same portion of code without being aware of each other: their local copies diverge in irreconcilable ways, and they have to undergo a painful process of analysis and coordination to produce a unique consistent version of that piece of code.

CloudStudio targets the shortcomings of traditional CM systems to facilitate collaborative development by abandoning the update-commit paradigm and by integrating an awareness system of what other developers are doing into the IDE. This way, developers using CloudStudio never have to update, and commit only very infrequently, while being constantly aware of potentially conflicting edits as they set in, before fixing them becomes too burdensome.

4.2 Configuration Management Model

CloudStudio stores the current base version of a project's code in a relational database hosted "in the cloud". The database table consists of the four attributes:

$$\langle \textsc{File}, \textsc{Line\#}, \textsc{Text}, \textsc{Owner} \rangle \,,$$

which respectively indicate a source file name, a line number in that file, the text appearing at that line number, and which users (if any) are editing that line.

The database stores the *base* version of the codebase with a tuple:

$$\langle f, k, l, \perp \rangle$$

for each line l in position k in a project file f, where $\perp$ denotes *base* versions. For example, if the fifth line of file "stack.e" contains the signature of routine *push* (v: *INTEGER*), the database will store the tuple $\langle$stack.e, 5, *push* (v: *INTEGER*), $\perp\rangle$. Whenever a user u changes the line in position k in a project file f into the string l', the database adds the tuple: $\langle f, k, l', u \rangle$, which records u's version of the line.

Since a tuple is added for every user who edits a line, we can search for *conflicts* by looking up tuples that only differ in the last component, with two values other than $\perp$. For example, if Claudia changes *push*'s argument type to *ANY*, and Stu makes *push* return a *BOOLEAN* to signal whether the operation was successful, there is a conflict signaled by the two tuples $\langle$stack.e, 5, *push* (v: *ANY*), Claudia$\rangle$ and $\langle$stack.e, 5, *push* (v: *INTEGER*): *BOOLEAN*, Stu$\rangle$.

Whenever a user u performs an explicit *commit*, the base version of the project is updated to reflect u's latest edits. That is, for every tuple $\tau = \langle f, k, l', u \rangle$ in the database *without conflicts*, CloudStudio discards every tuple $\langle f, k, l, \perp \rangle$ (for every l), and replaces τ with $\langle f, k, l', \perp \rangle$. If τ has conflicts, the base version of that line does not change. Every commit generates a new base version of the project in the database; the previous base version is purged from the database but it is stored in a back-end Subversion repository, which allows developers to roll back to older stable snapshots of the project and to re-populate the database with them.

If two users u_1, u_2 are working in *interweave* mode, CloudStudio stores their edits in the same tuples; that is, if either u_1 or u_2 changes the line in position k in file f into l', the database stores the tuple $\langle f, k, l', \{u_1, u_2\} \rangle$.[1] Correspondingly, conflicts may arise between u_1 and u_2's edits and somebody else's but not between u_1 and u_2. Also, a commit by either one of u_1 and u_2 has the same effect of updating the base version to coincide with u_1 and u_2's.

4.3 Awareness System

CloudStudio's awareness system extracts information from the configuration management database and displays it according to user preferences. The basic behavior is that the editor shows the current user's edits, and the base version of every line untouched since the last explicit commit. Each user retains ownership of her uncommitted changes; others can see them but not modify or commit them.

[1] The straightforward details of how this is implemented with relational schema are not discussed.

On top of this, CloudStudio provides options to see the changes introduced by other developers. Each company and project has its own rules. The CloudStudio vision carefully refrains from imposing a specific methodology or process model, but provides the means to support such choices. The current user can select any other developer u and choose to:

- Display all changes introduced by u;
- Display where u introduced changes but do not show them;
- Display only where u's changes generate conflicts;
- Do not display changes by u at all;
- Work in *interweave* mode with u.

The last option avoids the introduction of conflicts and allows developers to modify lines collaboratively, in a way similar to GoogleDocs but with a level of granularity and control suitable for software development.

5 Other CloudStudio Features

A CloudStudio prototype is freely available at `www.cloudstudio.ethz.ch/demo/`; since it is entirely web-based, using it does not require downloading any software. The implementation combines an editor written in Eiffel (automatically translated to JavaScript [4]) with other functionalities implemented in Java using Google Web Toolkit v. 2.3, and leverages a MySQL database back-end.

CloudStudio currently supports development in Eiffel, but its architecture is extensible to other programming languages such as C, C#, and Java. Besides the innovative configuration management and awareness system described in Section 4, CloudStudio offers the basic functionalities of traditional IDEs such as EiffelStudio or Eclipse: an editor with syntax checking, a class browser to navigate the project, and integration with the compiler. At the time of writing, the complete implementation of interweave editing is underway.

In continuity with our related work on formal verification centered around EVE, the Eiffel Verification Environment (`se.ethz.ch/research/eve/`), CloudStudio integrates verification tools to help developers improve software quality. It currently supports testing with the AutoTest framework [13], and formal correctness proofs with AutoProof [14]. AutoTest performs random testing of object-oriented programs with contracts, and it has proved extremely effective in detecting hundred of errors in production software; AutoProof provides a static verification environment similar to Spec#[1] but for Eiffel. Both tools are fully automatic and integrated with CloudStudio's CM system: testing and proving sessions work on the current view selected by CloudStudio users, which flexibly may or may not include concurrent edits by other developers (as described in Section 4).

6 Case Study

This section presents a case study that compares the performance of two-programmer teams using CloudStudio against traditional CM practices. The overall goal of the case study is to assess the usability of CloudStudio and its advantages for collaborative development over traditional IDEs and CM techniques.

6.1 Development Tasks

The case study included three program development tasks, two focused on refactoring and one on testing; all applications were written in Eiffel.

R1: Task R1 targets an application implementing a card game (the card deck and the game logic); the complete application includes 210 lines of code over 4 classes. Task R1 requires refactoring of three classes, and development of new functionalities by extending the refactored classes; the task is collaborative because the new functionalities must work with the classes after refactoring. Refactoring included: method and field renaming; enforcement of Eiffel coding standards (e.g., capitalization, comments); re-arrangement of methods in groups (marked by the **feature** Eiffel keyword) according to their functionalities; code extraction into a new class.

R2: Task R2 targets an application modeling a coffee vending machine; users of the application have basic options to select coffee, can pay and receive change. The application includes 230 lines of code over 3 classes. Task R2 is similar to R1 except that it targets the coffee machine application: R2 requires refactoring and development of new functionalities by extending the refactored classes.

T1: Task T1 targets the same coffee machine application as task R2. It requires development of new functionalities (namely, the option to add milk to the coffee, and the dispatch of different cup sizes) and writing of test cases that achieve 100% code coverage on the new code. Task T1 is also inherently collaborative as the development of new functionalities and of test cases occur concurrently, according to the concept of *test-driven pair programming* [6].

6.2 Subjects and Experimental Setup

The subjects used in the study were six PhD students from our research group. All of them are experienced Eiffel programmers who frequently develop with EiffelStudio and Subversion (SVN) as part of their PhD research; none of them had used CloudStudio before the study, had taken part in its development, or has much experience with collaborative development.

We randomly arranged the six subjects in three pairs: Team1, Team2, Team3. Team1 first performed task R1 with CloudStudio and then task T1 with EiffelStudio and SVN. Team2 first performed task R1 with EiffelStudio and SVN and then task T1 with CloudStudio. Team3 first performed task R1 with CloudStudio and then task R2 with EiffelStudio and SVN.

Each team performed its sessions according to the following protocol. The two team members sat at the opposite corners of a large table with their laptops connected to the network. Before beginning, the second author (henceforth "the experimenter") gave a brief (5 min.) introduction to CloudStudio to both

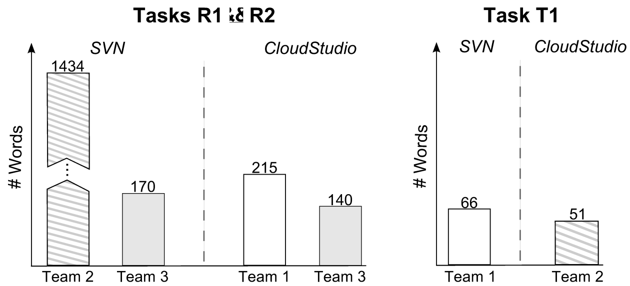

Fig. 2. Results of the case study (the scale is not uniform).

programmers at the same time, where he showed them how to log-in and the basics of the CM system without any reference to the development tasks. Then, he gave them a sheet of paper with a description of the task they had to perform (the second task was introduced only after completion of the first). The two programmers received identical instruction sheets and had to coordinate in order to split the work between them.

During the study nobody other than the experimenter and the two programmers was in the room. The programmers were only allowed to use instant messaging to communicate; their position in the room and the experimenter ensured that no other communication channel was available. The experimenter did not interfere with the programmers other than to clarify possible unclear points in the task description (but this was never necessary).

There was no time limit to complete the tasks: each session continued until the current task was completed (the experimenter checked completeness *a posteriori* by manual inspection of the codebase). After each session, the experimenter recorded the amount of communication between the two programmers—as the total number of words exchanged via instant messaging—and the overall time spent to complete the task.

6.3 Results

Figure 2 reports the amount of communication between programmers while performing the various tasks. While all participants are competent programmers, their speed and development style vary significantly; as a result, the random assignment formed heterogeneous groups which may not be directly comparable. The results in Figure 2, however, show a consistent advantage for teams using CloudStudio over teams using SVN: the difference is sometimes small (as for task T1), sometimes conspicuous (as for task R1 between Team2 and Team1); in all cases, CloudStudio required less communication for the same task than SVN, even if the study's programmers used it for the first time. Let us now describe the performance of the various teams in more detail.

Team1 delivered the best overall performance and was fluent both with SVN and with CloudStudio; the two programmers worked well together and required a limited amount of communication to synchronize properly. The comparison with Team2 on the same tasks suggests that using CloudStudio is beneficial: Team1 outperformed Team2 almost by an order of magnitude when using CloudStudio on task R1, whereas their performance became similar on task T1 where Team1 used SVN. It was clear that Team1 was overall faster than Team2, but the peculiarities of task R1 magnified the difference in favor of who could rely on better collaboration tools.

The programmers in **Team2** had the greatest communication problems in the study, as shown by their performance in task R1. The log of their message exchanges shows that they had to debate several points of disagreement about how to perform the refactorings, and that not being able to see in real-time what the other was doing (as it happened when working with SVN) exacerbated their disagreement and frustration.

Unlike the members of the other teams, the two programmers in **Team3** worked with wildly different speed, to the point that in both tasks R1 and R2 a programmer completed his part of the task when the other was still exploring the system and understanding the instructions. The overall performance of Team3 required little communication in all cases, but this is mostly a result of the fact that the different programmer speed forced a serialization between the two programmers; hence, synchronization was not a big issue because the development was not really collaborative and interactive.

We do not discuss in detail the *time* taken by programmers because the assignments emphasized correctness of the solution and did not pressure the teams for time. Anyway, and perhaps unsurprisingly, the overall time turned out to be correlated with the amount of communication, hence all the experimental data point to the same qualitative conclusions.

6.4 Discussion

A *post mortem* analysis of the instant messaging logs shows recurring patterns of communications between programmers. The initial part of every session starts with a discussion of the task, after which the two programmers negotiate a division of the labor and agree on some synchronization mechanism. During development with SVN, messages such as "Did you update your project?" and "I'm done with implementing X and have committed" are frequent. With CloudStudio, the same messages occurs much more sparingly, and some of the remaining instances can probably be attributed to the programmers' limited familiarity with CloudStudio and how it works (in fact, in some cases of redundant notification messages using CloudStudio, the recipient replied with sentences such as "Just go ahead, I can see your changes live").

After the case study, we asked the participants to complete a simple questionnaire about their experience and with requests for feedback. The participants unanimously appreciated CloudStudio mechanisms for the real-time visualization of other people's changes, and for the immediate display of conflicts. Disagreement existed on how severe a problem are merge conflicts in everyday's software development: four programmers consider it a serious hassle and appreciate better mechanisms to prevent or manage conflicts; the other two maintained that merge conflicts can be reduced to a minimum with a little coordination.

In all, the participants to the study tend to agree with our conclusions that CloudStudio offers valuable features for collaborative development and a more flexible paradigm of CM. The generalizability of our results is necessarily limited by the case study's scope and size, as well as by its reliance on specific development tasks that emphasize real-time collaboration but may affect only a limited part of large software projects. In this sense, the reaction of one of the programmers in our study to task R1 is instructive: he was initially skeptical and remarked that he "would never do refactoring while another programmer is implementing new functionalities"; after using CloudStudio, however, he acknowledged that, with the right tools, such tasks can indeed be performed in parallel.

7 Related Work

The need for new tools that properly address the problems of collaborative and distributed development is widely acknowledged [26].

Web-based IDEs such as Cloud9 [23], CodeRun Studio [24], and Codeanywhere [21] provide fairly standard development environments accessible with a browser. This alleviates the problem of ensuring that remote team members develop with the same environment, but it does not address the specific issues of concurrent collaborative development, and in particular conflicts. Projects such as Skywriter [20] and Collabode [5] offer concurrent code editing, similar to CloudStudio's interweave mode, which prevent conflicts but is not flexible enough to be applicable in all situations.

IBM's Jazz [22] is a framework for distributed development built on top of the popular Eclipse IDE. Jazz offers advanced communication and collaboration mechanisms, but they are still built around a conventional development model where remote programmers do not work concurrently on the same piece of code. Another relevant differences between the Jazz approach and CloudStudio's is that the former is not web-based, hence it requires that every team member has a suitably configured Eclipse local installation. Some Jazz extensions provide features specifically designed for collaborative development; for example, Jazz Sangam [25] supports project editing according to the roles of pair-programming. However, these extensions still rely on traditional CM mechanisms that require explicit synchronization with update and commit operations.

Awareness systems target some of the problems related to the standard "commit-update" approach to CM, and in particular the handling of conflicts. Palantir [17] is an awareness system that notifies about artifacts changed by other developers; in a recent study [18], Palantir was demonstrated to help detect and resolve conflicts earlier than with traditional CM. Crystal [3] is based on similar principles as Palantir, and it helps developers predict, and possibly prevent, conflicts. FASTDash [2] is instead based on the interactive visualization of changes introduced by team members. Awareness systems are valuable tools, but they should be closely integrated with IDEs and with the rest of CM to maximize their effectiveness. CloudStudio's awareness system fully integrates with the rest of the IDE: changes and conflicts are displayed in the editor and they can be included in the compilation and verification runs; it is also quite flexible, in that users can selectively include or ignore changes made by others. Additional significant differences between CloudStudio and other awareness systems derive from the fact that the latter work around traditional CM models, rather than revisiting them. Hence, developers using CloudStudio can visualize and inspect not only conflicts but also non-conflicting edits made by others, and truly concurrent editing is possible.

Tools such as Syde [7] and CollabVS [8] introduce new models of CM, where more abstract and flexible change analyses are possible. Syde, for example, works on abstract syntax trees of the code, and defines changes as abstract operations on trees. Wloka et al. [27] introduce a different approach, called "safe-commit", which automatically runs unit tests on the developer's changed copy of a project and determines if the local copy can be committed without introducing undesirable "semantic" changes to the codebase. All these new paradigms promise to significantly improve standard CM, but they still have to become central in the development of IDEs to impact development habits and processes; Syde, safe-commit, and CollabVS are instead implemented as extensions of traditional IDEs (namely, Eclipse and VisualStudio). Another shortcoming of these new paradigms is that they still are centered around the notion of conflict (and conflict resolution), whereas CloudStudio makes programmers aware also of non-conflicting changes to enable real-time collaboration. Finally, CloudStudio offers advanced code analysis tools in the form of the verification tools AutoTest and AutoProof, in addition to its novel CM and awareness system.

8 Conclusions

We have described a new paradigm of cloud-based software development, addressing the needs of modern distributed projects, and presented the first version of a supporting web-based tool called CloudStudio, freely available for experimentation.

As can be expected with such a novel approach, many questions remain open, providing both theoretical research challenges and practical engineering goals. We plan to extend the new paradigm of configuration management outlined above. Much work remains on the IDE, in particular more sophisticated display of the changes introduced by the developers. More collaboration tools are needed, in particular to support the new modes of code inspection made possible and desirable by the Internet [12]. We also intend to perform more extensive empirical evaluation of the effectiveness of the ideas and tools, both in an industrial setting and through systematic use of the tools in the multi-university distributed DOSE project and course on distributed and outsourcing software engineering course [16].

Acknowledgments

We thank Le Minh Duc, Alexandru Dima, and Alejandro Garcia for contributing to the prototype implementation of CloudStudio. Julian Tschannen and Yi (Jason) Wei contributed to the integration of the verification tools, and participated to the case study with Yu (Max) Pei, Marco Piccioni, Marco Trudel, Scott West. CloudStudio's startup funding through the Gebert-Ruf Stiftung is gratefully acknowledged.

References

1. M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the Spec# experience. *Commun. ACM*, 54(6):81–91, 2011.
2. J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson. FASTDash: a visual dashboard for fostering awareness in software teams. In *Proc. of CHI*, pages 1313–1322, 2007.
3. Y. Brun, R. Holmes, M. Ernst, and D. Notkin. Proactive detection of collaboration conflicts. In *Proc. of ESEC-FSE*, 2011.
4. A. Dima. Developing JavaScript applications in Eiffel. Master's thesis, ETH Zurich, 2011.
5. M. Goldman, G. Little, and R. C. Miller. Collabode: collaborative coding in the browser. In *Proceeding of the 4th international workshop on Cooperative and human aspects of software engineering*, CHASE '11, pages 65–68, New York, NY, USA, 2011. ACM.
6. M. Goldman and R. C. Miller. Test-driven roles for pair programming. In *Proc. of CHASE*, pages 13–20. ACM, 2010.
7. L. Hattori and M. Lanza. Syde: a tool for collaborative software development. In *Proc. of ICSE*, pages 235–238, 2010.
8. R. Hegde and P. Dewan. Connecting programming environments to support ad-hoc collaboration. In *Proc. of ASE*, pages 178–187, 2008.
9. J. Herbsleb and D. Moitra. Global software development. *IEEE Software*, 18(2), 2001.
10. H. Holmstrom, E. O. Conchuir, P. J. Agerfalk, and B. Fitzgerald. Global software development challenges: A case study on temporal, geographical and socio-cultural distance. In *Proc. of ICGSE*, pages 3–11, 2006.
11. B. Meyer. The unspoken revolution in software engineering. *IEEE Computer*, 39(1):121–124, 2006.
12. B. Meyer. Design and code reviews in the age of the Internet. In *Proc. of SEAFOOD*, volume 16 of *LNBIP*. Springer, 2009.
13. B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf. Programs that test themselves. *IEEE Software*, pages 22–24, 2009.
14. M. Nordio, C. Calcagno, B. Meyer, P. Müller, and J. Tschannen. Reasoning about Function Objects. In J. Vitek, editor, *TOOLS-EUROPE*, LNCS. Springer-Verlag, 2010.
15. M. Nordio, M. Joseph, B. Meyer, and A. Terekhov. Preface to SEAFOOD 2010. volume 54 of LNBIP, Springer, 2010.
16. M. Nordio, R. Mitin, and B. Meyer. Advanced hands-on training for distributed and outsourced software engineering. In *Proc. of ICSE*, 2010.

17. A. Sarma, Z. Noroozi, and A. van der Hoek. Palantir: raising awareness among configuration management workspaces. In *Proceedings of 25th International Conference on Software Engineering (ICSE)*, pages 444 – 454, 2003.
18. A. Sarma, D. Redmiles, and A. V. der Hoek. Palantir: Early detection of development conflicts arising from parallel code changes. *IEEE TSE*, 99, 2011.
19. J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. Usable verification of object-oriented programs by combining static and dynamic techniques. In *Proc. of SEFM*, LNCS. Springer, 2011. (To appear).
20. `http://mozillalabs.com/skywriter`. Skywriter.
21. `https://codeanywhere.net`. Codeanywhere.
22. `http://www-01.ibm.com/software/rational/jazz/`. Jazz.
23. `http://www.cloud9ide.com`. Cloud9 IDE.
24. `http://www.coderun.com`. CodeRun Studio.
25. J. Vijay, S. Devide, A. Meneely, C.-W. Ho, L. Williams, and M. Devetsikiotis. Jazz Sangam: A real-time tool for distributed pair programming on a team development platform. In *Proc. of IRCSE*, 2008.
26. J. Whitehead. Collaboration in software engineering: A roadmap. In *Proc. of FOSE*, pages 214–225.
27. J. Wloka, B. Ryder, F. Tip, and X. Ren. Safe-commit analysis to facilitate team software development. In *Proc. of ICSE*, pages 507–517, 2009.