Collaborative Debugging

H.-Christian Estler* Martin Nordio* Carlo A. Furia* Bertrand Meyer*,†
*Chair of Software Engineering, ETH Zurich, Zurich, Switzerland
firstname.lastname@inf.ethz.ch
†ITMO National Research University, St. Petersburg, Russia

***Abstract*—Debugging—the process of finding and correcting programming mistakes—faces too the challenges of distributed and collaborative development. The debugging tools commonly used by programmers are integrated into traditional development environments such as Eclipse or VisualStudio, and hence do not offer specific features for collaboration or remote shared usage. In this paper, we describe CDB, a debugging technique and integrated tool specifically designed to support effective collaboration among developers during shared debugging sessions. We also discuss the design and results of an empirical study aimed at identifying features that can ameliorate the effectiveness of collaborative debugging processes; and at evaluating the usefulness of our CDB collaborative debugging approach. The study suggests that CDB's collaboration features are often perceived as important for effective debugging; and can improve the overall debugging experience in collaborative settings.**

***Index Terms*—Distributed software development, Debugging, Empirical study**

I. Introduction

Debugging is the process of dealing with one of life's inevitables—programming mistakes. Always a cardinal part of the development process, its importance has prompted the construction of techniques and tools that can assist programmers to make them more productive at finding and fixing errors. Debugging techniques have typically been developed around traditional development practices, and debugging tools have become part of every integrated development environment (IDE) such as Eclipse or VisualStudio. This entails that they are essentially conceived as tools for individual usage, and hence may be a poor match for today's increasingly collaborative and distributed development processes.

This paper investigates the problem of deploying debugging techniques and tools in the context of collaborative development, where developers working on the same project cooperate with the goal of finding and correcting errors in their shared codebase. Debugging in collaborative environments is likely to feature as a useful activity in every team development practice, but is especially crucial—if not outright necessary—in the increasingly common *distributed* development settings: when developers of the same team may be located in physically distinct locations, possibly even in widely different time zones, the fine-grained coordination, required by collaboration on a highly interactive process such as debugging, becomes a real challenge. We offer three main contributions addressing the general problem of collaborative debugging, with an eye towards distributed settings.

Debuggers—such as the GNU debugger or the Microsoft VisualStudio debugger—are normally integrated in IDEs without specific support for shared usage. Programmers working on the same project synchronize indirectly through shared repositories managed using tools such as Subversion or Git, but their debugging sessions are individual and cannot benefit from collaboration unless they are sitting at the same desk—something hardly possible in a distributed setting. The first contribution of this paper is a debugging technique, called CDB, designed for remote collaboration. CDB supports multiple programmers, each one sitting at her desk, sharing a common execution of the program under debugging. Every programmer can add or remove breakpoints, inspect variables, and navigate the code, with the others aware of each other's actions in real-time. We implemented this debugging technique leveraging the features of CloudStudio, the web-based IDE for collaborative development we introduced in previous work [7]. Section III discusses our collaborative debugging technique and how it can be used in practice. CloudStudio and CDB are available online[1].

Evaluating the effectiveness of a complex and ultimately human-driven process such as debugging is a challenging problem even in traditional single-user sessions. Parnin and Orso [16], for example, provided evidence that the traditional protocols used to evaluate fault localization techniques often misrepresent or overstate their real effectiveness when used within actual debugging sessions. In collaborative settings, designing sound empirical evaluations becomes even more challenging, as more variables (such as the number of people collaborating) must be controlled and accounted for. The second contribution of this paper is the design of an empirical study of collaborative debugging in possibly distributed settings. Our experimental protocol accommodates analysis focused on the interaction model—for example, as in CDB vs. using traditional processes. We describe the protocol in Section IV.

Our third contribution is an actual empirical study of collaborative debugging, following the experimental design just outlined. The study involved 38 participants, performing 10 debugging tasks of various difficulty using either our collaborative debugging technique CDB or a standard remote-desktop application to follow and interact with their debugging partners. We split the students in different groups to achieve a good trade-off between statistical significance and generalizability of the results; in particular, we experimented with different interaction models and included both pairs and

[1] http://se.inf.ethz.ch/research/cloudstudio/

triples of programmers sharing the same debugging session. Section V describes the details of the study, whereas Section VI analyzes potential threats to validity. The study's most significant findings are: the two features that are generally perceived the most important for collaborative debugging are:

- being able to independently browse the code under debug;
- and add variables to watch.

There is no evidently preferred mode of control, but debugging with CDB provides for collaborative debugging with a more uniform level of involvement, as well as for debugging processes perceived as more efficient and generally preferred for collaborative tasks.

Before presenting our contributions, Section II gives an overview of the challenges of using debugging tools collaboratively, in a settings where programmers may be displaced at different locations and interact only remotely.

II. Collaborative Debugging in Distributed Teams

Let us introduce our two fine programmers Pippo and Binha. Pippo lives in Italy and has recently joined our team of developers working on a large Java application. His current assignment involves using a binary tree implementation written a few weeks ago by Binha—who belongs to the Brazilian development unit.

To get started and understand the binary tree's API, Pippo writes the client code in class `Client` shown at the bottom of Figure 2. His code creates an instance `t` of class `BinaryTree` and populates it with a few nodes of class `Node` storing integer values. The rest of Figure 2 outlines the essential parts of the `BinaryTree` and `Node` classes written by Binha.

It does not take long before Pippo realizes that something's wrong with the binary tree implementation. Since he has access to the whole codebase, Pippo may simply debug the implementation on his own. This approach has, however, some evident drawbacks. Since Pippo is not familiar with the details of the binary tree implementation, he is likely to be slow at debugging it. Worse, he may not be aware of the other clients' usage requirements, and his fixes may negatively affect them without him realizing it. Finally, touching base with Binha is probably advisable in any case, just to avoid that the two of them introduce conflicting changes which will require a later painful merge process.

So Pippo asks Binha to help him debug the problems he is facing. Since they have no specific tool support for collaborative debugging, the best they can do is using a remote-desktop application: Pippo's computer is running the debugger within Eclipse; and Binha is connected remotely, sees whatever Pippo sees in his computer screen, and can text him with comments or requests for actions.

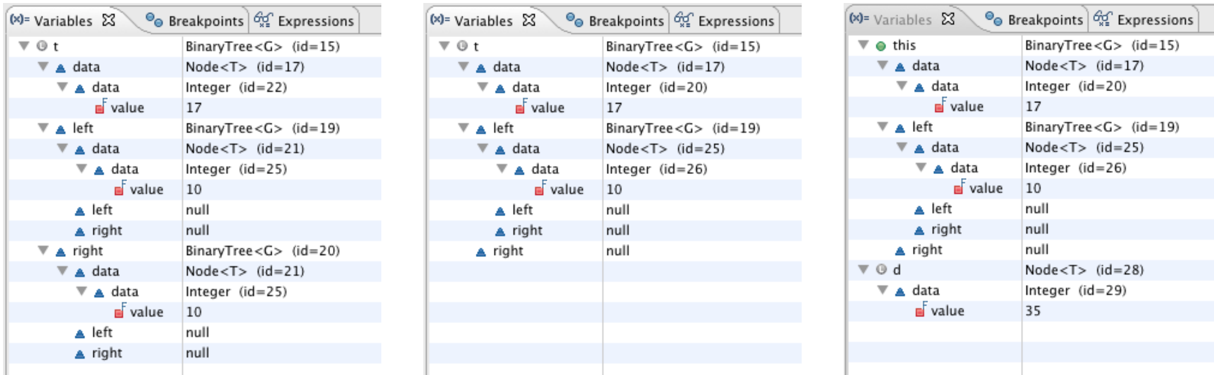
To demonstrate the problem, Pippo adds breakpoints at lines 36, 37, and 39 and starts the debugger. When execution stops at the first breakpoints (line 36), the debugger displays a pane with the variables in scope as in Figure 1a. This shows that the node with value 10 has been incorrectly inserted twice in the tree. Binha realizes that the problem is the conditional at line 15, which should be exercised only if the previous condition on line 10 fails. Binha suggests to change the **`if`** on line 15 into an **`else if`**. She cannot do the change directly and her view is limited to the code currently displayed in Pippo's IDE window; instead, she explains the problem to Pippo via voice chat and asks him to deploy the change.

```
1  class BinaryTree<G extends Comparable<G>> {
2    Node<G> data;
3    BinaryTree<G> left, right;
4
5    public BinaryTree (Node<G> root) {
6      data = root; left = null; right = null;
7    }
8
9    public void insert (Node<G> d) {
10     if (d.lessThan(data)) {
11       if (left == null) {
12         left = new BinaryTree<G>(d);
13       } else { left.insert(d); }
14     }
15     if (d.greaterThan(data)) {
16       if (right == null) {
17         right = new BinaryTree<G>(d);
18       } else { right.insert(d); }
19     }
20   }
21   public void traverse(int i) { ... }
22 }
23
24 class Node<T extends Comparable<T>> {
25   T data;
26   public Node(T d) { data = d; }
27   public boolean lessThan(Node<T> n) { ... }
28   public boolean greaterThan(Node<T> n) { ... }
29 }
30
31 class Client {
32   public void main() {
33     BinaryTree<Integer> t =
34         new BinaryTree<>(new Node<Integer>(17));
35     t.insert(new Node<Integer>(10));
36     t.insert(new Node<Integer>(30));
37     t.insert(new Node<Integer>(35));
38     t.insert(new Node<Integer>(12));
39     t.insert(new Node<Integer>(21));
40     t.traverse(0);
41   }
42 }
```

Fig. 2: A Java binary tree implementation.

After fixing as suggested by Binha, Pippo re-starts the debugger. Upon reaching the breakpoint at line 36, it now shows the state in Figure 1b, which looks fine. Pippo issues a step-over command, which continues execution until the next breakpoint at line 37, that is it inserts a node with value 30. It is clear, however, that there is still a problem, as the insertion does not actually change the state of the tree.

After coordinating with Binha again over voice chat, Pippo issues a step-into command, which causes the debugger—now at the second breakpoint at line 37—to show the code executed by the call `t.insert(35)` which inserts a node with value 35. The debugger shows that the condition `d.lessThan(data)` on line 10 evaluates to false; the next condition `d.greaterThan (data)` on line 15 is also false given the state of Figure 1c. Thus, the problem is with the implementation of `lessThan` or



(a) Before fixing the first bug. (b) After fixing the first bug. (c) State at line 15 after fixing the first bug.

Fig. 1: Debugging the binary tree example.

greaterThan:

```
43 public boolean lessThan(Node<T> n)
44 {   int last = this.data.compareTo(n.data);
45     return last < 0 ? true : false; }
46
47 public boolean greaterThan(Node<T> n)
48 {   int last = this.data.compareTo(n.data);
49     return last > 0 ? false : true; }
```

Evaluating the expressions `this.data.value` and `d.data.value` in the current debugging context gives the values 17 and 35, and variable `last` in library method `greaterThan` correspondingly evaluates to 1 because `35 = d.data.value > this.data.value = 17`. The conditional expression on line 49, however, incorrectly returns **false**. Pippo and Binha agree that a suitable change is switching the returned values on line 49 which becomes `return last > 0 ? true : false`.

III. Collaborative Debugging with CDB

Even if the debugging session described in Section II eventually succeeded, it showed that using a remote-desktop application to collaborate has several shortcomings. In fact, achieving effective collaboration during shared debugging sessions seems to have conflicting requirements. On the one hand, the programmers involved should be able to *share* a common debugging session; ideally, each should be able to modify the code or interact with the debugger directly, without need to describe actions in speech or via chat and request someone else to carry them out. At the same time, each programmer should also be able to perform *independent* activities in parallel, such as browsing parts of the project or testing the effects of small changes to the code, without interfering with or depending on what his colleagues are doing in their IDEs. Remote-desktop applications provide sharing, but subject to a strict discipline where one or more clients have access to a master machine. The clients' view on the master machine are limited to what is displayed on its screen at any time, and there is no simple way to coordinate or even to switch the role of master with one of the clients.

Based on these preliminary observations, which the empirical study of Section V will corroborate, we designed CDB, a collaborative debugger that facilitates remote coordination.

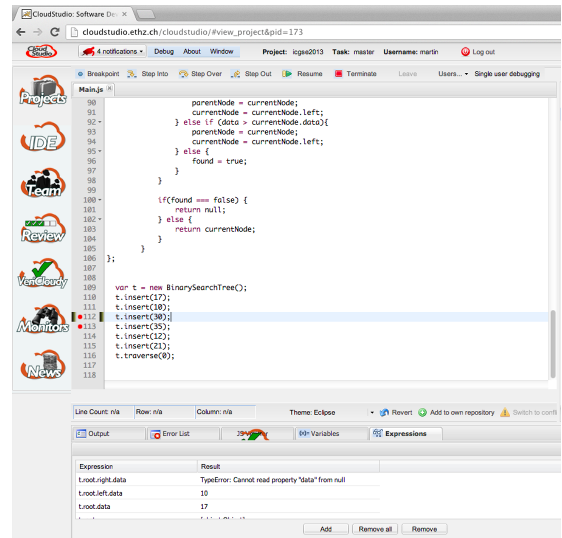


Fig. 3: CDB running the example of Figure 2.

CDB is integrated within the CloudStudio web-based IDE, which we developed in related work [7]. Users of CDB log-in to the CloudStudio server using any web browser. Whenever they select a common project to work on, they can start *shared* debugging sessions using CDB. A shared session has a unique thread of execution on the project under debugging; all participating users observe the program state throughout the shared session, for example by displaying the values of certain variables. Figure 3 shows a screenshot of CDB running on the example of Section II.

CDB offers the standard commands to control debugging sessions: *add* or *remove* breakpoints (where execution pauses); *step-over* a call (execute it as an atomic statement); *step-into* a call (execute the next of its constituent steps); *step-out* of a call (switch back to the higher level after a step-into); and *resume* or *terminate* the whole session. All collaborating users see the effect of any issued commands on their shared session.

A crucial issue for usability is achieving a suitable balance of flexibility and discipline in how to control debugging

sessions: flexibility for users to issue commands without requiring explicit coordination—thus reducing the communication overhead—and discipline to avoid haphazard debugging sessions—thus reducing the impact of conflicting strategies. To this end, CDB offers two control modes, which achieve different trade-offs between flexibility and discipline. While all users are allowed to insert or remove breakpoints in either mode, the other commands are managed differently:

Master mode: a single user is allowed to issue the commands step-over, step-into, step-out, resume, and terminate; the other users observe the execution controlled by the master. At any time during the session, the master can switch roles with one of other users. This is often useful when the execution reaches parts of the code the master is not familiar with; he may then decide to become an observer and tap in a more knowledgeable collaborator, who takes the lead without having to start over with a new debugging session.

Peer mode: all connected users are allowed to issue any of the commands. To avoid some extreme situations where conflicting commands are issued whose net effect is void (for example, a step-into followed by a step-out shortly afterward), whenever a user X issues a command, CDB rejects any new command issued by any other user Y within a couple of seconds. This retains some coordination discipline by implicitly requiring that Y waits at least until X concludes a sequence of arguably closely related commands.

Debugging with CDB offers additional perks that derive from being integrated within the CloudStudio IDE. While the debugging session itself is shared, each user retains exclusive control on what happens in her IDE. As the session unravels, she may perform useful activities in parallel, such as browsing relevant part of the codebase, adding provisional changes and sharing them with the others using CloudStudio's configuration management and real-time awareness system [7]. In summary, users collaborate on the shared debugging session but may work asynchronously to get a clearer picture of what is going on and consequently direct the debugging process based on better informed decisions.

Debugging with CDB. CDB's features can improve the collaborative debugging experience of the example discussed in Section II over using standard remote-desktop applications; in particular, synchronization can happen more efficiently leveraging CDB's control modes. For example, Pippo initiates the debugging session acting as master; Binha can still add breakpoints to stop the execution at crucial points. She can also navigate the source code of the project independent of what Pippo does in his IDE window, to acquire information without blocking others. Pippo can easily hand control over to Binha whenever execution reaches parts that she knows better. And changes to the code can also be performed by either programmer, with the other aware of them in real-time thanks to CloudStudio's configuration management mechanisms.

IV. Empirical Study: Experimental Design

The overview of Section II highlighted some issues that are likely to surface when performing debugging in collaborative distributed settings; and the presentation of our collaborative debugger CDB in Section III suggested an approach that may improve the effectiveness of the debugging experience in such settings. The rest of the paper describes an empirical study aimed at evaluating issues of and approaches to collaborative debugging.

The study's main goal is testing whether the CDB approach addresses the relevant issues, and whether it brings tangible benefits. The main research questions are correspondingly as follows (the rest of the section gives a characterization of "effective collaboration"):

RQ1: Which debugger features (e.g., adding breakpoints) are critical for an effective collaboration in debugging sessions?

RQ2: What is the relevance of different control policies (e.g., a single programmer always in control) for an effective collaboration in debugging sessions?

RQ3: How do the two debugging experiences, using remote-desktop vs. using CDB, compare?

Let us now discuss how we collected evidence to answer these general questions.

Assignments and tasks. We evaluate programmers working on two assignments: the *List* assignment and the *Tree* assignment. The two assignments have comparable size and complexity, and differ only in the kind of data structure they target—respectively linked lists and binary search trees. Each assignment comes with 100–200 lines of source code written in JavaScript (the example of Section II is a simplified Java variant of the *Tree* assignment). We concocted the assignments with the goal of having codebases sufficiently simple, so that programmers can find their ways through them in the limited time allotted by the experiments, but also not entirely trivial.

An assignment consists of five *tasks*; each task comes with a test case that reveals an error of the assignment's data structure; the task's goal is fixing the error through debugging. The participants are also given a written description of the tasks, including detailed instructions and a short tutorial describing the tools and how to use them. They are also allowed to ask for clarification to the supervisor.

The first task in each batch is simpler than the other four, and the programmer performance on it is not evaluated. Since tasks are executed in order, the first task serves as a warm-up: programmers get a chance to acquire some familiarity with the codebase and with the debugging tools at their disposal. This increases the similarity of the experimental setup with real debugging scenarios, where it is likely that programmers already have some knowledge of the codebase and of the tools. It may also reduce the impact of different participants to the study having different previous experience with the tools.

Debugging tools and scenarios. To make the debugging experiences using remote-desktop vs. using our CDB comparable, we need to set up debugging environments that

are as similar as possible except for the debugging tools. For example, comparing remote-desktop using the Eclipse debugger (as in Section II) to CDB integrated in CloudStudio would make little sense, given that a widely-used and mature IDE such as Eclipse makes for an overall quite different user experience than the research prototype CloudStudio. Instead, we set up two usage scenarios for CloudStudio: CD and SS. Under scenario CD (for "CDB" debugging), developers use CDB exactly as described in Section III, with the shared debugging session, the various control modes, as well as all other features of CloudStudio. Under scenario SS (for "shared-screen" debugging), one master developer uses CloudStudio on her machine as if it were a single-user browser; the other developers participating to the debugging session connect to the master machine using a remote-desktop application, can only watch what is displayed on the master machine and interact with the master via text chat, Skype, or voice (the last option is obviously possible when they are in the same room) as described in Section II. In this way, the features offered by the bare IDE are the same, whereas the collaboration means are quite different in the two scenarios.

We randomly split the participants in three groups, using different combinations of CD and SS:

G1: programmers in this group debug following scenario SS. Half of the group works on *List* and half works on *Tree*.

G2: programmers in this group debug following scenario CD. Half of the group works on *List* and half works on *Tree*.

G3: programmers in this group debug following both scenarios CD and SS. This is organized in two subgroups, according to which scenario they follow first:

- **G3.A:** programmers first work on an assignment under SS and then work on the other assignment under CD. Half of the group works first on *List* and then on *Tree*; the other half works first on *Tree* and then on *List*.
- **G3.B:** programmers first work on an assignment under CD and then work on the other assignment under SS. Half of the group works first on *List* and then on *Tree*; the other half works first on *Tree* and then on *List*.

Group G3 makes it possible to evaluate if the user experience changes when programmers have a chance to try both scenarios CD and SS and to compare them. The split of G3 into G3.A and G3.B helps control for the influence of getting experience with debugging under one scenario on debugging under the other scenario: since both CD and SS use the same basic CloudStudio IDE, the performance on the second assignment may improve just as a result of becoming familiar with the IDE.

Team allocation. We split programmers in each group in debugging *teams*. Members of the same team interact following scenario SS or CD. Our study does not measure the effect of distribution on debugging performance, even though we had a mixture of teams whose members were in the same room, in the same building, in different locations in the same city, and even in different countries.

To evaluate the effect of increasing levels of collaboration, each group includes debugging teams of different size. In our experiments, we focused on 2-programmer and 3-programmer teams: 2-programmer teams are the baseline for collaborative debugging, whereas 3-programmer teams demonstrate whether more collaboration is achievable with sustainable overhead. Future studies focusing on collaboration may experiment with teams of even larger size, after addressing the criticalities shown by our study.

Previous experience. To control for previous experience of the participants, we ask them to rank on a 1–5 scale their experience with programming in general, with JavaScript programming, and with interactive debugging. We also report whether they had already experienced collaborative debugging of any kind before the study.

A. Critical Debugger Features: RQ1

Research question RQ1 looks for critical features of debuggers used collaboratively. After a team has completed its assignments, we ask each of its members to rank on a 1–5 scale the importance of the following features: adding and removing breakpoints, adding and removing monitored variables and expressions, and browsing the codebase independent of others. An additional free-text question asks to mention any other feature that they consider important.

Answers to questions about the same feature are not directly comparable between teams working under different scenarios, who have or don't have experienced that feature. In fact, questions for teams following SS are phrased as "How much did you miss feature X?", whereas the corresponding questions for teams following CD are phrased as "How useful was feature X?". Instead, we can directly compare the same questions about SS between teams in group G1 and in group G3, as well as about CD between teams in group G2 and in group G3. We also analyze *correlations* between questions about different features for the same group.

B. Control Policies: RQ2

Research question RQ2 studies the impact of the control policies allowed or enforced in the different sessions. After a team has completed its assignments, we ask each of its members to rank on a 1–5 scale: the importance that everyone on the team can issue commands (e.g., step-into) to the debugger; the level of active involvement in the debugging exercise; the degree of control achieved on the debugger. Again, questions were phrased differently for sessions following SS and sessions following CD: questions for teams following SS are phrased as "How much did you miss that everyone can issue commands?", whereas the corresponding questions for teams following CD are phrased as "How useful was it that everyone can issue commands?". We also ask to rank on a 1–5 scale: how often each member issued commands (CD scenarios) or asked the person in charge to issue commands (SS scenarios), where 1 denotes "Never" and 5 denotes "More than 12 times".

When collecting and comparing data about control policies, it is especially important to account for the role each pro-

grammer had in the debugging session and for the number of people involved. To this end, we partition the answers to the questions according to whether the respondent was the master: in SS scenarios, the master is the person who is in control of the IDE; in CD scenarios using master mode, the master is the person who is allowed to issue commands.

For the CD scenarios, we also run sessions where CDB operates in peer mode (see Section III), where there is no single master but all team members can issue commands at any time. Correspondingly, the questionnaires for sessions using CD also included questions about the usefulness of the master or peer modes as control policies.

C. Debugging Experience: RQ3

Research question RQ3 draws a comparison between the debugging experiences using remote-desktop (as in scenario SS) and using CDB (as in scenario CD). After a team has completed its assignments, we ask each of its members to rank on a 1–5 scale the perceived efficiency of debugging under the assigned scenario. For teams in group G3, who experienced both debugging scenarios, we also ask which one they preferred. Finally, we measure the overall time to complete the tasks (not including the first warm-up task); and the number of tasks successfully completed within a limit of 60 minutes.

V. Empirical Study: Execution and Results

We performed the empirical study described in Section IV with 38 participants: 19 bachelor's students, 10 master's students, and 9 professional programmers, spread across Italy, Switzerland, and Croatia. We distributed the participants into groups G1, G2, and G3 and into 2-person and 3-person teams as shown in Table I. We decided the number of components in each group, but the specific assignment of people to groups was random.

GROUP:	**G1**	**G2**	**G3.A**	**G3.B**
SCENARIO:	SS only	CD only	SS, then CD	CD, then SS
# 2-person teams	4	1	4	4
# 3-person teams	1	2	–	1
# teams debugging List	3	2	5	4
# teams debugging Tree	2	1	4	5

TABLE I: Setup of groups and assignments. Participants used shared screen (SS), collaborative debugging (CD), or both techniques to debug one or both of the assignments *List* or *Tree*. Precisely, teams in **G1** and **G2** debugged one of either *List* or *Tree*; teams in **G3.A** debugged both assignments using SS for *List* and then CD for *Tree*; teams in **G3.B** debugged both assignments using CD for *List* and then SS for *Tree*.

As shown in Figure 4, the previous experience of the participants spans multiple levels. In particular, nearly all participants have significant programming experience; most of them have repeated debugging experience; about half of them have a little experience with JavaScript, collaborative debugging, or both. The mosaic plot in Figure 4 also shows that the distribution of experience is comparable in the various groups, namely those working only with SS, only with CD, and with both SS and CD (in any order).

A. Critical Debugger Features: RQ1

Which features are most useful during collaborative debugging? The boxplot of Figure 5 and the corresponding Table II report the results of the questions targeting RQ1 concerning debugging sessions under scenario SS (using no specific support for collaboration other than remote desktop). Answers are ranked on a 1–5 scale, with 1 denoting "feature not missed" and 5 denoting "feature much missed". The possibility of browsing the code independent of other teammates is the most missed feature, followed by being able to add expressions and variables to be monitored.

The boxplot of Figure 6 and Table III summarize the answers to the corresponding questions concerning debugging under scenario CD (using our collaborative debugger CDB). Answers on a 1–5 scale now denote features from "considered not useful" to "considered very useful". Browsing code still is a very popular feature, and so it the possibility of adding variables. Adding or removing breakpoints, which everyone can do at any time with CDB, is not considered particularly useful, nor is often missed in the SS scenario. This is probably due to the fact that, even when only one programmer can change the breakpoints, asking for a breakpoint change requires little communication (essentially, a location).

The participants did not report any other generic feature in the specific open-answer questions. In all, browsing the code and modifying the variables and expressions monitored by the debugger emerge as critical features for effective collaborative debugging.

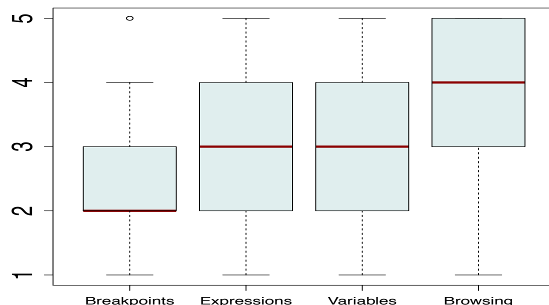


Fig. 5: Features missed during SS debugging, on a 1–5 scale.

FEATURE	#	min	median	max	mean	σ
Breakpoints	30	1	2	5	2.5	1.17
Expressions	30	1	3	5	3.1	1.18
Variables	30	1	3	5	3.33	1.3
Browsing code	25	1	4	5	3.6	1.35

TABLE II: Features missed during SS debugging (# is the number of answers, σ is the standard error).

Impact of being the master. While in debugging scenarios CD with CDB all programmers in the team may browse the code and take control of the debugger, the master is strictly

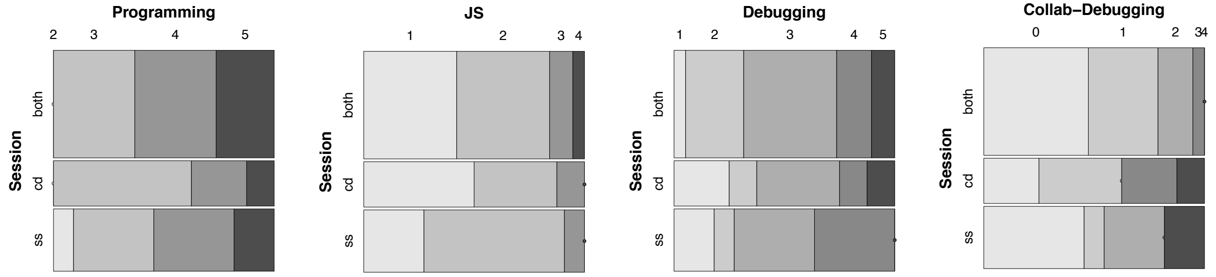


Fig. 4: Previous experience with programming in general (left), JavaScript programming (center left), debugging (center right), and collaborative debugging (right) amongst the study participants. All measures on a 1–5 scale, except collaborative debugging which is on a 0–4 scale.

FEATURE	n	min	median	max	mean	σ
Breakpoints	27	2	3	5	3.48	1.12
Expressions	27	2	4	5	4.15	0.99
Variables	27	2	5	5	4.19	1.04
Browsing code	27	2	5	5	4.33	1.11

TABLE III: Features considered useful during CD debugging (n is the number of answers, σ is the standard error).

fixed during the SS debugging sessions, where only the person sitting in front of the computer running the debugger can browse, add variables, expressions, and breakpoints, as well as start and stop the debugger.

FEATURE	$\#M$	$\#O$	U	p
Breakpoints	11	19	82	0.33
Expressions	11	19	109	0.86
Variables	11	19	103	0.96
Browsing	9	16	74	0.91

TABLE IV: Significance test to determine if the role during SS debugging sessions impacts how much a feature is missed. $\#M$ is the number of programmers in the role of master, $\#O$ the number of all other programmers.

To understand whether being the master affects the perception of which features are important, we performed statistical significance tests comparing the answers about missing features given by masters vs. the other programmers in the team. Since the data may not be normally distributed and is scaled ordinal but not continuous, we use an independent two-group Mann-Whitney U test, with null hypothesis H_0: $\mathbb{P}(M < O) = \mathbb{P}(O < M)$, where M measures the answers given by programmers in the role of master, and O the answers given by all the others. Table IV shows the results. Since $p \geq 33\%$ for every feature, we do not reject H_0: there is no evidence that being the master affects the perception of which features are critical for collaborative debugging.

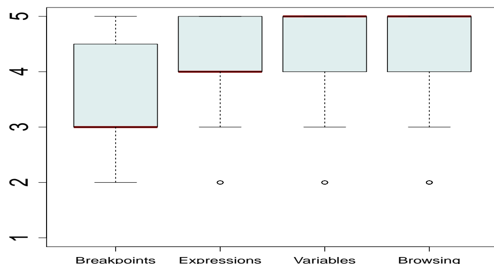


Fig. 6: Features considered useful during CD debugging, on a 1–5 scale.

Impact of group allocation. While the answers to the questions for SS scenarios (“Which features did you miss?”) and the corresponding questions for CD scenarios (“Which features did you find useful?”) are not directly comparable, it is interesting to see whether taking part in both debugging scenarios affects the perception of either set of questions.

Feature	G1	G3	U_{SS}	p_{SS}	G2	G3	U_{CD}	p_{CD}
Breakpoints	11	19	127	0.33	8	19	74.5	0.96
Expressions	11	19	134	0.20	8	19	57.5	0.30
Variables	11	19	92	0.61	8	19	102.5	0.13
Browsing	11	14	108	0.08	8	19	88	0.46

TABLE V: Significance test to determine if having experienced both debugging scenarios impacts the importance attributed to features. Columns G1, G2, and G3 list the number of programmers in the corresponding groups.

To this end, we performed statistical significance tests comparing the answers about the debugging scenario SS (respectively, CD) coming from people in group G1 (respectively, G2) and in group G3. Again, the nature of the data suggests a U-test with the obvious null hypothesis. Table V shows the results, with the left-hand half of the table about scenario SS and the right-hand half about scenario CD. Since $p \geq 13\%$ for every feature, we do not reject the null hypothesis: there is no evidence that having experienced both debugging scenarios affects the perception of which features are critical for collaborative debugging.

Correlation analysis. Tables VI and VII show the correlation coefficients between the variables measuring the background of participants and the features they considered useful; as usual, Table VI first shows the data about sessions under

scenario SS and then Table VII the data about sessions under scenario CD.

	P	D	Js	Cd	K	E	V
D	**0.58**						
Js	0.03	0.20					
Cd	0.18	0.36	0.35				
K	0.14	−0.14	**0.40**	0.19			
E	0.15	0.20	0.18	0.03	**0.52**		
V	0.15	0.23	0.22	0.00	**0.36**	**0.60**	
B	**0.67**	**0.59**	0.10	0.12	0.17	**0.49**	**0.62**

TABLE VI: Spearman correlations between variables for sessions under scenario SS. Numbers in bold are statistically significant at a 5% level or better. The variables measure the experience with Programming, with Debugging, with JavaScript, with Collaborative debugging, as well as how much adding/removing breaKpoints, Expressions, Variables, and independent Browsing was missed.

	P	D	Js	Cd	K	E	V
D	**0.54**						
Js	0.12	0.18					
Cd	−0.16	0.25	0.09				
K	−0.27	−0.25	−0.29	−0.11			
E	−0.29	0.12	−0.17	−0.04	**0.40**		
V	0.30	0.16	0.05	−0.33	0.31	**0.38**	
B	0.31	0.07	−0.15	**−0.50**	**0.40**	0.30	**0.71**

TABLE VII: Spearman correlations between variables for sessions under scenario CD. Numbers in bold are statistically significant at a 5% level or better. The variable names are as in Table VI.

Looking only at the statistically significant correlations, we find the obvious one between programming experience and debugging experience: it is hard to progress in programming without plenty of debugging involved. Valuing independent browsing correlates with valuing adding variables; after all, these are both consistently ranked as the two most important features. The correlation between adding variables and adding expressions is also significant, probably since the latter can be seen as a generalization of the former. The other significant correlations occur only in one of the two scenarios (SS or CD) and fail straightforward interpretations. For example, the correlation between JavaScript programming and valuing adding breakpoints is reasonable—to the extent that a better understanding of the program makes for a more effective control of the debugging sessions—but it is not clear why it is only significant for SS debugging scenarios. Such open points are good material for future studies.

B. Control Policies: RQ2

The second research question studies the impact of control policies, and in particular who is the master and how this role can change.

Amount of commands. Table VIII shows the amount of commands the programmers requested to the master (when in SS scenarios) or performed themselves (when in CD scenarios). The data is very similar under the two scenarios, and in fact a U test does not provide any evidence otherwise ($U = 248$ and $p = 0.86$).

SCENARIO	#	min	median	max	mean	σ
SS	30	1	2	4	2.26	1.15
CD	30	1	2	4	2.56	1.19

TABLE VIII: Amount of commands requested to the master by others (SS scenarios) or performed by any programmer (CD scenarios), on a 1–5 scale.

A correlation analysis shows only one significant correlation: for sessions under scenario SS, the amount of commands requested to the master significantly correlates ($\rho = 0.53$, $p < 0.05$) with the general programming experience of those issuing the requests. This confirms the intuition that general programming experience tends to be an indicator of the capability of being actively involved in debugging processes.

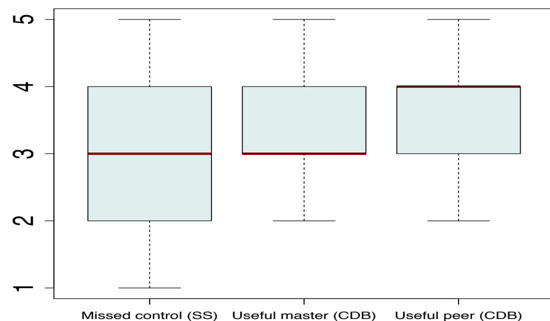


Fig. 7: For SS debugging sessions: importance that everybody can issue commands. For CD debugging sessions: usefulness of the master and peer modes (see Section III).

Impact of control policies. The leftmost bar in Figure 7 shows how much programmers participating in SS debugging sessions missed that everyone can directly issue commands to the debugger. A high variability range suggests that lacking direct control is perceived differently by different people. An obvious guess would be that control is missed the most by who does not have it, that is programmers other than the master in SS scenarios. A U test does not, however, give any support to this guess ($U = 81.5$ and $p = 0.32$). Therefore, the explanation may have more to do with general attitudes towards collaboration; but further investigation is needed to answer conclusively.

Regarding sessions under the CD collaborative scenario, the participants slightly preferred the peer control policy, where everyone can issue commands at any time. The difference is, however, small and not statistically significant: a 2-group Wilcoxon signed rank test (the data is paired) gives $V = 97$ and $p = 0.37$.

Involvement and control. Another set of questions asked what was the level of involvement during the debugging sessions. To see to what extent involvement is influenced by the role and by the debugging scenario (SS vs. CD), Table IX breaks down the data into: SS sessions for programmers

	#	min	median	max	mean	σ
SS: master	11	3	4	5	4.56	0.69
SS: other	19	2	3	5	3.16	0.90
CD	27	2	4	5	3.89	0.69

TABLE IX: Involvement of programmers: during SS sessions when in master role; during SS sessions when not in another role; during CD sessions. Data on a 1–5 scale.

SCENARIO	#	min	median	max	mean	σ
SS	30	1	3	4	2.57	0.86
CD	27	1	4	5	3.27	1.10

TABLE X: Process efficiency in each scenario.

in the master role; SS sessions for programmers in another non-master role; CD sessions, where the role of master is interchangeable or collective. With statistical significance, the master is typically more involved than the others in SS debugging sessions ($U = 183$ and $p < 10^{-3}$); and the latter are typically less involved than anyone participating in CD debugging sessions ($U = 155.5$ and $p = 0.02$). The difference between programmers in CD sessions and masters in SS sessions is, instead, borderline statistically significant ($U = 199.5$ and $p = 0.08$). In all, there is evidence that a collaborative approach such as that offered by CDB makes for a more uniform distribution of involvement, which is an important goal in collaborative activities.

C. Debugging Experience: RQ3

The third research questions looks at the overall collaborative debugging experience with remote-desktop (SS scenarios) and with our tool CDB (CD scenarios).

Among the 19 programmers who tried both SS and CD (group G3), 17 (or 89%) claimed to *prefer* the experience with CDB over the interaction using remote-desktop. A related question asked to rate the *efficiency* of the process in each of the two scenarios. Table X shows the answers. The difference is still in favor of debugging using CDB, with good statistical significance: $U = 230$ and $p = 0.0037$.

In all, the data gives us some confidence that, even if other variables are not greatly affected, CDB is a step in the right direction of supporting collaborative debugging.

VI. Threats to Validity

Internal validity. A few shortcomings in the execution of the empirical study (Section V) constitute potential threats to interval validity. We could not always enforce a strict time limit to complete each assignment, mainly due to misunderstandings arising with the geographically distributed programmers who took part to the study. While most sessions completed within one hour, a few went on for longer than two hours. We minimized the impact of this problem by using neither time nor tasks completed as measures in the evaluation, even though we collected this data. Another inconsistency occurred with three two-person teams working under the SS scenario: members of the same team could not use different computers, so they simply sat at the same terminal working together. In these situations, we still applied the protocol that limited their communication to verbal (as if they were connected by Skype) and forbade them from pointing to the screen, or sharing the control of the keyboard or any other input device. This should have limited the impact of the threats to validity in this case. A general limitation of the study originates from the usage of CloudStudio as web-based IDE. Since CloudStudio still is a research prototype, it lacks advanced IDE functionalities, and is not immune from bugs (in particular, the real-time synchronization mechanisms may transiently lose responsiveness due to imperfect load balancing). These may affect how programmers rate their debugging experience. However, we performed all debugging sessions (SS and CD) using CloudStudio, so as to have a common baseline and only evaluate differences relative to it.

External validity. Though we designed the empirical study's tasks to resemble real-world debugging scenarios, the code examples were necessarily limited in complexity and size, and the bugs to be fixed were introduced on purpose. Such limitations apply to all "laboratory" studies of programming activities, as there is no simple recipe to guarantee that in-the-small tasks are indicative of real-world programming. While debugging is a complex multi-faceted process, which we cannot expect to understand with a single empirical study, in-the-small studies such as ours can help single out important factors, which can then be assessed more thoroughly in follow-up larger-scale studies.

VII. Related Work

As far as we know, this paper's contribution is novel, both in presenting a new approach and supporting tool for collaborative debugging where distributed users share a common debugging session in real-time; and in performing an empirical evaluation of debugging in collaborative settings. This section, describes the most relevant related work in two areas: tools for distributed software development (DSD) and the features they offer for debugging; and empirical studies of DSD.

A. Tools for Distributed Software Development

The arsenal of tools for distributed software development is quickly expanding, and also includes mature commercial tools such as Microsoft Team Foundation [13] and IBM Jazz [3]. These tools support various aspects of the collaboration between developers, such as sharing code and documentation in the early implementation phases. They are also well integrated with the corresponding IDEs—VisualStudio and Eclipse in the case of Team Foundation and Jazz. The VisualStudio debugger offers some support for collaboration: one developer can freeze a debugging session running on her machine and transfer its state to another computer running VisualStudio; there, it can be restored and continue with another user. To be able to take over a debugging session, the two user must have access to the same codebase. IBM Jazz offers a similar debugging functionality.

Such approaches to "transferable" debugging are useful but not truly collaborative, as they offer no support for the real-time sharing that may be needed to have faster and more

directed interactions. Our CDB tool also supports transfer of control during debugging sessions, but in real-time without requiring that a session be frozen, transferred, and restored.

With IDEs such as CodeRun [5], Cloud9 [4], and Collabode [9], tools for software development have been following the general trend of moving to the web. These IDEs offer functionalities similar to traditional IDEs, but are usable without installation through a web-browser. Even if the code is stored on a shared server and accessed transparently, every user works on a logically different copy of the code, through standard configuration management practices. Cloud9 and Collabode also supports real-time collaboration: multiple users simultaneously edit the same piece of code, as if they were working on a GoogleDoc shared document. None of these web-based IDEs offer collaborative debugging functionalities.

JS Bin [19] is a web-based collaborative tool for developing JavaScript programs, which supports a form of collaborative debugging. The collaboration is achieved by publishing a URL where the current debugging session is shown, providing an experience similar to using remote-desktop applications (as illustrated in Section II and referred to as SS scenario in the empirical study of Section V). DebugLive [20] offers similar functionalities for passive collaboration. With both tools, only one user is in charge of browsing the code, adding or removing break points, and issuing other commands to the debugger; the other participants can only watch and give suggestions.

B. Empirical Studies of Collaborative and DSD

Distributed software development has become a standard practice in today's software industry, one with many challenges [2], [12], [11]: differences in time zones and cultural backgrounds, increased difficulties of performing requirements engineering, project management, and API design, just to mention a few. Some of these challenges have been investigated empirically. For example, the effect of time zones on various phases of development [10], [6], [14]; the relation between development processes and distribution [8]; the effects on productivity and quality [17], [1]; the usage of contracts for API design [15]; and the impact of geographic dispersion on quality metrics [18].

To our knowledge, there is no study about the collaborative aspects of debugging processes and tools such as those discussed in the present paper.

VIII. Conclusions

This paper presented CDB, a debugging technique and integrated tool to support effective collaborative debugging. We evaluated collaborative debugging—in general and with the CDB approach—through an empirical study whose main findings are:

- The two most critical features useful to improve collaborative debugging are: the possibility of browsing code independent of collaborators, and of changing the watched variables in the running debugging sessions.
- Collaborative debugging tools, such as CDB, that allow collaborators to easily switch the role of who is in control of a debugging session achieve more involvement of all debugging session participants.
- Collaborative debugging with CDB is perceived as more effective than the alternative of sharing a single-user debugging session with only indirect interactions possible.

As future work, we plan to extend CDB with more functionalities such as a stack call, to collect more data on its usage with real-world applications, and to examine its potential for impact on how collaborative debugging is performed.

Acknowledgments. Thanks to Rand Nezha and Mert Tufekci for contributing to the implementation of CDB, and to all the students who participated in the case study. CloudStudio's startup funding through the Gebert-Ruf Stiftung is gratefully acknowledged. Work partially supported by ERC grant # 291389.

References

[1] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy. Does Distributed Development Affect Software Quality? An Empirical Case Study of Windows Vista. In *ICSE*, pages 518–528. IEEE, 2009.

[2] E. Carmel. *Global software teams: collaborating across borders and time zones*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[3] L.-T. Cheng, C. R. de Souza, S. Hupfer, J. Patterson, and S. Ross. Building collaboration into ides. *Queue*, 1(9):40–50, December 2003.

[4] Cloud9 IDE. http://www.cloud9ide.com.

[5] CodeRun Studio. http://www.coderun.com.

[6] J. A. Espinosa, N. Nan, and E. Carmel. Do Gradations of Time Zone Separation Make a Difference in Performance? A First Laboratory Study. In *ICGSE*, pages 12–22. IEEE, 2007.

[7] H.-C. Estler, M. Nordio, C. A. Furia, and B. Meyer. Unifying configuration management with merge conflict detection and awareness systems. In *ASWEC*. IEEE, 2013. To appear.

[8] H.-C. Estler, M. Nordio, C. A. Furia, B. Meyer, and J. Schneider. Agile vs. structured distributed software development: A case study. In *7th International Conference on Global Software Engineering*. IEEE, 2012.

[9] M. Goldman, G. Little, and R. C. Miller. Collabode: Collaborative coding in the browser. In *Proceeding of CHASE '11*, pages 65–68, New York, NY, USA, 2011. ACM.

[10] J. D. Herbsleb, A. Mockus, T. A. Finholt, and R. E. Grinter. Distance, dependencies, and delay in a global collaboration. In *Proceedings of CSCW '00*, pages 319–328, New York, NY, USA, 2000. ACM.

[11] 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 *ICGSE '06*, 2006.

[12] B. Meyer. The unspoken revolution in software engineering. *IEEE Computer*, 39(1):121–124, 2006.

[13] Microsoft Team Foundation. http://www.microsoft.com/visualstudio/en-us/products/2010-editions/team-foundation-server/overview, 2012.

[14] M. Nordio, H.-C. Estler, B. Meyer, J. Tschannen, C. Ghezzi, and E. D. Nitto. How do distribution and time zones affect software development? a case study on communication. In *ICGSE*, Los Alamitos, CA, USA, 2011. IEEE.

[15] M. Nordio, R. Mitin, B. Meyer, C. Ghezzi, E. D. Nitto, and G. Tamburelli. The Role of Contracts in Distributed Development. In *SEAFOOD*, volume 35 of *LNBIP*, Berlin, Heidelberg, 2009. Springer-Verlag.

[16] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *ISSTA*, pages 199–209. ACM, 2011.

[17] N. Ramasubbu and R. Balan. Globally Distributed Software Development Project Performance: An Empirical Analysis. In *ESEC/FSE*, pages 125–134. ACM, 2007.

[18] N. Ramasubbu, M. Cataldo, R. K. Balan, and J. D. Herbsleb. Configuring Global Software Teams: A Multi-Company Analysis of Project Productivity, Quality, and Profits. In *ICSE*, pages 261–270. ACM, 2011.

[19] http://jsbin.com/. Jsbin.

[20] http://www.breakpoints.com/. Debuglive.