Experiments in Distributed Side-By-Side Software Development

Prasun Dewan
Department of Computer Science
University of North Carolina
Chapel Hill, USA
dewan@unc.edu

Puneet Agrawal
Gautam Shroff
Tata Consultancy Services, India
puneet.a@tcs.com
gautam.shroff@tcs.com

Rajesh Hegde
Microsoft Research
Microsoft
Redmond, USA
rajesh.hegde@microsoft.com

***Abstract*—In distributed side-by-side software development, a pair of distributed team members are assigned a single task and allowed to (a) work concurrently on two different computers and (b) see each others' displays. They can control when they communicate with each other, view each others' actions, and input concurrently. To understand how this control is exerted in practice, we have performed experiments at two different organizations, Microsoft Research and Tata Consultancy Services, which involved about forty six person hours of distributed side-by-side development. The experimental tasks were typical of the kind carried out at these organizations. A mix of qualitative, quantitative, and visualization analysis shows that (a) distribution and conflicting changes are not an issue; (b) developers use the unique capabilities provided by distributed side-by-side software development; and (c) the exact usage depends on several factors such as the collaboration task, developers, and software-development abstraction and environment.**

Keywords-pair programming; conflicts; visual programming

I. INTRODUCTION

Complex software must be developed collaboratively. However, Brooks[1] observed that adding more people to a software team can result in a disproportionate increase in coordination cost, thereby reducing the productivity of the individual programmer. If the coordination cost is really an issue, then distributing the team should further aggravate this problem and radically co-locating it in a single war-room should reduce it. Two independent studies have found that this is indeed the case - the productivity of co-located teams was higher than that of distributed teams[2]; and the productivity of teams radically-co-located in a single "war-room" was greater than that of co-located ones working from different cubicles[3].

The above works have assumed that programmers are assigned separate tasks, with no coupling between their workspaces. This is not the case with pair programming, wherein two developers work together on the same problem, taking turns in providing input. Some studies of pair programming claim that it offers not only faster task completion times, but also, after taking into account the cost of fixing bugs, better productivity[4, 5]; thus seemingly contradicting the observation of Brooks, which as mentioned above, did not consider workspace coupling. Another study [6] has found that pair design is even more effective than pair programming, requiring fewer person hours to produce the correct answer in comparison to independent design.

Recent work has proposed a variation of pair programming, called side-by-side programming, wherein two programmers, sitting next to each other and using different workstations, work together on the same task[7]. A recent study has shown that, in comparison to pair programming, side-by-side programming offers significantly lower completion times[8].

Previous work on side-by-side software development, however, leaves several questions answered. How can it be applied to the distributed case? How effective is such development – in particular, does it impose a performance penalty, and is it easy to use as the co-located case? It allows users to control when they communicate with each other, view each others' actions, and input concurrently. In what ways is this control exerted in practice, and on what factors does it depend? In particular, given the differences reported in pair programming and pair design, what is the effect of the level of abstraction on the side-by-side collaboration modes chosen by the developers? What kind of visualizations can show these differences in different side-by-side software development sessions? Given that side-by-side software development can allow developers to work independently, can it lead to conflicting changes?

A previous workshop paper[9] by us on this subject presented two systems for supporting distributed side-by-side development. It also provided some preliminary experience with the system based on five experiments. This is a follow-up paper, focusing, as the title indicates, on experiments. It provides a more in-depth evaluation of the systems in several ways. First, it describes results of six more experiments, which were performed after the workshop paper was published. Second, it provides visual and quantitative analyses, which were missing in the workshop paper. Finally, its qualitative analysis considers several additional factors such as the level of abstraction used by the developers.

The research described here was carried out at two different organizations, Microsoft and Tata Consultancy Services (TCS), during summer visits to these organizations by the first author. The two case studies are interesting because of the great differences in the software development environments used in the two groups involved in this project. The Microsoft group

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8295
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8295

used a traditional environment in which the application-development language (C#) was compiled, the application code was explicitly divided into multiple files, and a desktop programming environment and version control system was used to manipulate and store these files. The TCS group, on the other hand, developed web-based J2EE applications. It used a weakly-typed interpretive, proprietary web development environment called InstantApps[10]. Developers did not explicitly break the artifacts they created (such as JavaScript files and form definitions) into files and check them in and out – instead they simply pushed their changes to and loaded them from the InstantApps meta-data transactional repository server. The environment provided multiple levels of abstraction to develop an application. As a result, it allowed us to determine the influence of the level of abstraction on the nature of side-by-side software development. Because of the variety in the InstantApps abstractions, the bulk of our experiments were carried out at TCS.

The rest of the paper is organized as follows. Section II describes previous work on which our research is built. Section III motivates and describes our experimental tasks, and present a visual, quantitative, and qualitative analysis of the experiments. Finally, Section IV summarizes our contributions and gives directions for future work.

II. PREVIOUS WORK

Various forms of collaborative software development can be distinguished by the degree of coupling among the workspaces of the members of a software team.

In the traditional programming model studied by Brooks, no coupling occurs directly among these workspaces – all sharing occurs when a workspace is committed to a shared repository. Higher-degrees of coupling support continuous collaboration (an extension of the idea of continuous coordination[11]), wherein programmers can be aware of ongoing edits made by their team members. Several systems have been developed to support such collaboration including Jazz[12], Palantir[13], and CollabVS[14].

It is possible to use these systems to support an infinite range of workspace couplings. However, to the best of our knowledge, only four degrees of workspace coupling have been studied so far, which, in order of coupling degree, are traditional no-coupling, continuous conflict resolution, side-by-side programming, and pair programming.

In continuous conflict resolution, programmers work concurrently on different tasks, coupling their workspaces only to identify and resolve conflicts. A higher coupling occurs in side-by-side programming, wherein two programmers, using different workstations, work on the same task. The highest form of coupling occurs in pair programming, which is a component of extreme programming. Here two programmers, using a single (logical) workstation, work on the same task and provide input serially. The developer providing input is called the driver, and the other programmer is called the navigator. The developers take turns in being the driver/navigator.

In this paper, we use the term "coupling" broadly to refer to both (a) updates to local workspaces in response to edits made by others, and (b) awareness of actions of others. In the CSCW literature, the former is sometimes referred to as coupling and the latter as awareness[15].

The non-traditional coupling modes have been studied to some degree in lab/field studies. Two lab studies have shown that continuous conflict resolution provides a "stitch in time" by preventing (the commitment of) costly conflicts that would be detected at test or usage time in traditional programming [13, 14]. The recent study[8] mentioned earlier showed that, in comparison to pair programming, side-by-side programming offers significantly lower task completion times while slightly reducing the understanding developers have of code written by their partners. It also showed that developers who liked working together on a single task preferred side-by-side programming to pair programming. Previous work has not characterized the ways in which side-by-side programming is used, how appropriate it is for design tasks, and how it can be distributed - issues addressed by our work.

There have been several, often contradictory, studies of pair programming. One study of student programmers implementing class-assignments finds that pair programming takes more person hours but results in about 15% fewer bugs[4, 5]. The additional time taken depends on whether the students have previously done pair programming together. The first time a pair works with each other, they take about 50% more time, and subsequently, they take about 15% more time. The difference is explained by the time needed to "jell together" the first time. Assuming certain times for fixing and detecting bugs, the study claims that pair programming actually increases the productivity of an individual programmer.

The study also found that in the pair programming case (a) 80% of programmers felt higher satisfaction, (b) more alternatives were explored and fewer lines written, and (c) there was more team building as programmers were involved with each other and enjoyed celebrating project-completion together.

At least one other study seems to confirm the above work [16]. It differed from the previous study in two main ways. First, the programmers were from industry rather than a university. Second they did a single 45-minute lab exercise with their partners rather than multiple class assignments. The study found that, in comparison to uncoupled programming, pair-programming took about 50% more person hours but resulted in about 50% better readability, 30% better functionality, 75% more enjoyment, and 75% more confidence in solution.

Not all studies of pair programming have been as positive. Nawrocki and Wojciehowski[17] found pair programming often took about twice as many person hours, though the pair-programming times showed less variance. Ratcliffe and Robertson[18] found that programmers with high (self-reported) skills did not like being paired with those with low skills.

The above studies have been empirical. Based on the results of some of these studies and numbers reported in the literature on various factors influencing project costs such as the time it takes to write a line of code and fix a defect, and the cost of

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8295
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8295

missing a deadline, Muller and Pandberg[19] characterize cases where pair-programming is more cost effective and those in which independent programming is more economic.

All of the pair-programming studies mentioned above have assumed that the programmers are co-located. Some researchers have explored a distributed version of pair programming, where two remote programmers using different workstations view a single logical workspace. The shared logical workspace is created by coupling either (a) the screens of the users using a generic desktop sharing system, or (b) the edit buffers and other components of the semantic state of the software development environment of the programmers. The former is slower, but the latter requires the developers to manually synchronize their views. A study comparing distributed and co-located pair programming[20] has found that distance does not matter. This is an interesting result, as several studies of traditional programming have found that distance reduces productivity[2], though one shows that this is not always the case[21].

Motivated by these results, we developed a distributed analogue of side-by-software development, reported in our workshop paper[9]. Each developer in the pair interacts with two computers (Figure 1) – one primary computer to act as the driver of his subtask, and an awareness computer to act as the navigator for the partner's subtask. Thus, each programmer interacts with the windows displayed on his/her primary computer, and each awareness computer shows the screen of the partner's primary computer. The developers use the phone to talk to each other. No video channel is established between them.

Figure 1. Distributed side-by-side ingteraction model. The display of the primary computer of the left (right) developer, A(B), is shown on the awareness computer of the right (left) developer.

Because of the differences in the development environments used in the two organizations, we created two different implementations of this interaction model. Both implementations use desktop sharing systems to display the screens of primary computers on the corresponding remote awareness computers. Moreover, in both implementations, the two remote primary computers are also coupled. This coupling is provided in the Microsoft environment by the file system, and in the TCS environment by the InstantApps server. More details about the motivation and operation of the implementation are given in the workshop paper[9].

III. EVALUATION

A. *Evaluation Goals and Comparison Perils*

The goal of this research was to determine the appropriateness of distributed side-by-side collaboration in the two groups involved in the research, and industry, in general. Hence we decided to choose for our study the kind of tasks given to these groups, and use actual members of the groups as subjects. As we see below, this decision had an impact on several aspects of our evaluation.

Our initial approach was to run experiments that compare distributed side-by-side, pair and traditional uncoupled software development. For the evaluation metric, we chose task completion times. We decided to perform N rounds of experiments, where each round involved three pairs of developers, each of which performed three pairs of tasks using the three modes of development, respectively. To illustrate, consider a round with three developer pairs, (P_{11}, P_{12}), (P_{21}, P_{22}), (P_{31}, P_{32}), and three task pairs, (T_{11}, T_{12}), (T_{21}, T_{22}), (T_{31}, T_{32}). First, P_{11} and P_{12} used traditional uncoupled development to independently complete tasks T_{11} and T_{12}, respectively, while P_{21} and P_{22} (P_{31} and P_{32}) worked collaboratively on both of these tasks, using distributed pair (side-by-side) software development. We measured the time it took to complete each of the two tasks in all three cases. As this time could depend on the pair, we next changed the task pair to (T_{21}, T_{22}), and made each pair of developers implement it using a collaboration mode they had not used before; and finally, we changed it to (T_{31}, T_{32}), and repeated the process. At the end of this round, each pair had used each distributed collaboration mode. The developers performed the exercises as they would any task – at their own desks with no time limits.

After the first round of experiments, we found that the task completion times did not follow any trend. A close analysis of the recordings showed us that this was partly because some pairs, unlike others, made small, "silly," mistakes that cost them a significant amount of time. For example, one pair at TCS made the mistake of assuming that the field they had called "service provider" was actually named "solution provider," and it took them more than half an hour (one person hour) to discover this mistake as the environment does not check that a referenced field is actually defined. This problem of weak typing did not occur in the Microsoft experiments, but even in these experiments we saw such small but costly errors. In one of these exercises, a complex recursive routine had an error that the first author (watching the experiment remotely) immediately noticed but which took the pair more than forty five minutes (one and a half person hour) of testing and inspection to discover and fix. Anyone who has programmed can relate to such mistakes, and the time taken to fix them can add up quickly.

We could have run a large number of experiments to try and cancel the impact of these mistakes. However, we found that an even more fundamental reason for the variance in task completion times was that most of the tasks, especially those at TCS, involved user-interface design. This was consistent with the actual tasks assigned to the groups. The different pairs took different amounts of time to discuss and craft the user-interfaces, and there was no objective standard to evaluate the quality of the user-interfaces. Specifying the details of the user-interface for them would have been inconsistent with our goal of making the tasks realistic for the involved groups.

Yet another cause for the variation at TCS was that our requirement documents for the tasks were modeled after, and

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8295
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8295

were typically parts, of actual requirement documents given by customers. These were somewhat ambiguous, which resulted in different pairs implementing different functionalities.

This variation of task completion times is consistent with the tremendous variation in the results of some of the studies comparing pair and traditional programming, surveyed in Section II.

On further reflecting about this problem, we realized that there was another valid objective way to evaluate (distributed) side-by-side development, which relies on the fact that it subsumes traditional and pair development, in that users can choose to ignore the primary or awareness computers, respectively. Thus, instead of trying to determine what style of collaboration yielded the best productivity, we could simply study how the subjects worked together. If they primarily did traditional or pair development, then side-by-development offers no value, otherwise it does. This approach is a special case of evaluating a technology, based not on how it improves productivity, but to what extent its various aspects are actually used, assuming only useful features are used.

Our goal, then, became characterizing the coupling used at each time moment in various (distributed) side-by-side sessions, and determining the factors on it depended – in particular the software development abstraction. This required us to formally define the notion of coupling and abstraction.

B. *Coupling and Abstraction Characterization*

As mentioned before, side-by-side collaboration allows developers to control when they communicate with each other, view each others' actions, and input concurrently. This flexibility allows the definition of a wide range of couplings at each moment of time:

- *Driven 1(2)*: User 1(2) works on his primary computer, at least one of them talks, and user 2(1) does not work. Thus, this is a pair programming mode in which user 1 (2) is the driver and user 2 (1) is the navigator.
- *Discussion*: User 1 or 2 talks, and neither of them works.
- *Concurrent-coupled*: Both user 1 and 2 work on their primary computers, and at least one of them talks.
- *Concurrent-uncoupled* Both user 1 and 2 work on their primary computers, and neither of them talks.
- *Solo 1(2)*: User 1(2) works on his primary computer, user 2(1) does not, and neither of them talks.
- *Thinking*: Neither user 1 or 2 works or talks.

Driven, discussion, solo, and thinking can also occur in pair development; and solo, concurrent and thinking can also occur in traditional development (Figure 2). The distinguishing aspect of distributed side-by-side development is the ability to support and switch among all of these modes.

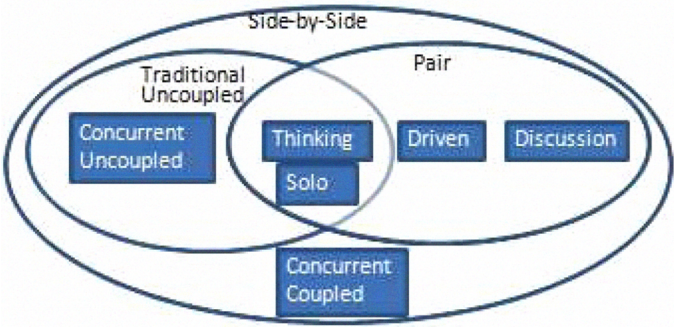

Figure 2. Coupling modes supporte by traditional indivodual, pair, and side-by-side development.

As mentioned above, pair design and programming yielded different results in some experiments. The software development mechanisms used at Microsoft and TCS allowed us to study abstractions of four different levels, where the lower the level of the abstraction, the more the details given to the computer:

1. Understanding requirements: The developers read paper/electronic copies of the requirements. They do not make any edits. This is the highest abstraction used.

2. Visual data: A visual language is used to define the data schema and the user interface for entering the data. This abstraction was used only in TCS, and corresponded to specifying forms, tabs, and menus.

3. Visual algorithms: A visual language is used to specify semantic constraints on application data. This abstraction was used only in TCS, and involved specifying workflows, roles, and access permissions.

4. Text coding: A procedural language is used to develop software. In TCS, the language was JavaScript, and at Microsoft, it was C#.

C. *Exercises*

Our next step was to determine the coupling-abstraction relationship in practice. As mentioned above, after the first round, we focused only on distributed side-by-development. As we could not use a comparative approach, we decide to focus on task diversity in our experiments.

At Microsoft, the exercises were maintenance problems involving improvements to existing research software implemented using the model-view-controller framework:

1. Scanning: A scanner that assumes a fixed amount of space between tokens is changed to support a variable amount of space between the tokens.
2. Shape groups: A drawing editor is extended to support grouping of shapes into nested shape groups.

These two exercises were performed by two different distributed pairs.

The TCS exercises involved writing several Web applications from scratch:

1. Pet Store: Owners arrange specific pets into categories.

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8295
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8295

2. Help Desk: An employee fills a ticket to request help.
3. Conference review system: Authors submit papers, which are evaluated by reviewers and then selected by a conference chairman.
4. Work request tracking: A customer makes a request for some work assigned to a worker by a team lead.
5. Purchase order: A customer orders a set of items from a specific vendor.
6. Stock count: A retailer manages the count of various items in the store.

The TCS experiments involved six distributed pairs. Each pair implemented two of these applications.

Each pair also did an unrecorded warm-up exercise to get used to distributed side-by-side development and "jell" with each other, which was not considered in the evaluation.

Thus, fourteen two-person experiments were carried out, two at Microsoft, and twelve at TCS. As mentioned before, the bulk of the experiments were carried at TCS because of the variety in the InstantApps abstractions. The session of each developer was recorded using screen and audio capture software. Due to technical difficulties, we could not use the recordings of one of the pairs – hence we effectively used twelve pairs of recordings, whose total length was about 46 hours.

We developed requirements documents for all exercises. The first author wrote the requirements documents for the Microsoft tasks. As mentioned before, at TCS, our requirement documents were modeled after, and were typically parts, of actual requirement documents given by customers.

Our study participants were employees of TCS/Microsoft. They included five women and eleven men in the age range 20-33. The Microsoft subjects included the last author (12 years industrial experience) and three summer interns. The TCS subjects were all full-time employees with 1-4 years industrial experience.

D. Annotated Recordings and Intersections

As an initial cut at characterizing the coupling-abstraction relationship in the twelve sessions, we divided the collaboration time into discrete units, and for each unit, we manually recorded (a) whether the developer talked/worked during that time, and (b) the abstraction used by the developer.

The time granularity used was 1 minute. If the user talked during any portion of a minute, he or she was considered to have talked during that minute. A developer was considered to have worked during a minute if the screen changed during that period. The first abstraction used by the developer in the minute was the one that was noted. The coder had to use the context of the conversation to determine the abstractions used.

As this is a novel way of characterizing collaboration sessions, we could not rely on a standard approach to choosing this granularity. Naturally, the finer the granularity, the more precise the measurements, but the more the effort required to manually code the units, and the more the chance of making errors in noting the passage of time and making the correct annotation. As we had about 46 hours of recordings, a 1 minute granularity yielded about 3000 annotations. A smaller time unit would have been hard to keep track of and resulted in, we believe, an inordinate amount of effort. Instead of using a fixed length granularity, we considered noting the time of each event. However, this would have further increased the annotation effort and error, as there were, typically, several speech utterances and display changes in a minute. We also considered automatic techniques for detecting speech and display changes, but found that reliably eliminating background noise is still a research topic.

We wrote programs that intersected the talking/working annotations for each collaborative session to derive the joint coupling at each moment of time.

E. Quantitative Analysis

We used these data to compute aggregate statistics regarding the abstraction/coupling relationship. In such statistics, user 1 and user 2 have no meaning, as they are bound to different users in each experiment. Therefore, when presenting such data, we combine solo1 (driver 1) and solo 2 (driver 2).

Table 1 shows, for each abstraction level, the percentage of time spent in each coupling mode (on average, across all sessions). For example, the 0.45 entry in the cell corresponding to Requirements row and Discussion column indicates that, on average, 45 percentage of the total time spent on requirements was in the discussion coupling mode. Thus, in each row, the numbers add up to 1 (100 per cent).

Table 1 does not indicate the percentage of time spent on each abstraction. The Total column of Table 2 gives this data. For example, the 12.9 entry for the Requirements row and the Total column indicates that on average 12.9 per cent of the total time was spent on requirements. Given this column, it is possible to determine the fraction of the total time spent on a particular abstraction and coupling. For example, the fraction of the total time spent in the discussion mode and requirements abstraction is 12.9 * 0.45 = 5.82. This information is given in the columns of Table 2. The bottom row gives the fraction of the total time spent in the various coupling modes across all abstractions. Each value in this row is a column sum.

Several conclusions can be drawn from these tables about the average time spent on various couplings and abstractions.

Table 1 shows that the degree to which a coupling mode was used depended on the abstraction. Requirements involved only one percent use of the two uncoupled modes – solo and concurrent-uncoupled. The next lower abstraction, visual data, involved eleven percent use of these modes, while the two lower-level abstractions, visual algorithm and text coding, had much greater use – forty two and twenty five percent, respectively. These findings are consistent with the intuition that (a) a higher-level abstraction allows more time to be spent on decisions, and less in execution of these decisions, and (b) when two people are working together, decisions are usually made collaboratively.

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8295
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8295

Table 1 Percentage of time spent in each coupling mode in each abstraction

	Solo	Conc-Uncpld	Discussion	Driven	Conc-Coupled	Thinking
Requirements	0.00	0.01	0.45	0.37	0.08	0.07
Visual Data	0.08	0.11	0.04	0.36	0.40	0.00
Visual Alg.	0.15	0.27	0.05	0.21	0.32	0.00
Text Coding	0.02	0.23	0.04	0.22	0.47	0.00

Table 2 Percentage of total time spent in each coupling mode and abstraction

	Solo	Conc-Uncpld	Discussion	Driven	Conc-Coupled	Thinking	Total
Requirements	0.03	1.17	5.82	4.82	1.11	0.93	12.9
Visual Data	3.93	5.68	2.19	18.26	20.31	0.28	50.6
Visual Alg.	2.53	4.71	0.78	3.64	5.54	0.03	17.3
Text Coding	0.46	4.50	0.68	4.36	9.22	0.00	19.2
Total	7.0	15.1	9.4	31.1	36.2	1.2	

The bottom row of Table 2 shows the relative use of the various coupling modes. Concurrent coupled and driven modes were by far the dominant modes, accounting for 36 and 31 percent of the time, respectively. The next most used mode was concurrent uncoupled, accounting for 15 percent of the time. 9 percent of the time was used for discussion, and 7 percent for solo development. An insignificant amount of time was spent in (pure) thinking. Thus, together, the exercises involved a significant use of all modes except the pure thinking mode.

As Figure 2 indicates, concurrent-coupled is unique to side-by-side development. The large fraction of the time (thirty six per cent) spent in this mode seems to contradict studies that have found that, typically, a person cannot perform two foreground activities (e.g. talking and driving in unknown terrain) simultaneously. As we show below in the section on qualitative analysis, there are two explanations for our results. One is that the developers had so much awareness of each other's activities, that they could easily perform their own activity and discuss the task of their partners. The second is that developers were often interleaving their working and talking, and the one minute granularity could not distinguish between interleaving and true concurrency. The distributed side-by-side interaction model made such rapid interleaving possible, as both developers' complete work was visible at all times.

The fact that the developers used one of the concurrent modes for about half the time is also fairly surprising, as it indicates they were able to dynamically break their task into components, and did not simply resort to pair development.

Table 1 also shows that all abstractions involved a significant use of all modes. The one exception was requirements analysis, in which ninety one percent of the time the developers used the coupling modes provided by pair development (Figure 2) – the concurrent-coupled (uncoupled) mode was used eight (one percent) of the time.

The above observations provide a new argument to motivate (distributed) side-by-side development in certain situations. Given a single task to implement, a pair of developers does not simply revert to traditional or pair development. Instead, they prefer the unique ability of side-by-side programming to switch between existing modes and the additional mode supported by it. This finding motivates distributed tools that allow developers to easily switch between coupled and uncoupled and concurrent and serial work.

One of the issues in collaboration is the amount of time a person spends on some activity before switching to another activity or being interrupted by a collaborator. This time gives an idea of how often a user switches context, and maybe loses important state of the previous context. We defined a related metric, atomic segment, which is specific to side by development. It is the time period during which neither programmer changes his/her abstraction and the coupling mode. We found that the average length of the atomic segments for different modes were remarkably close to each other, lying in the range: 2.3 to 2.75 minutes.

The aggregate data presented here does not describe the influence on the coupling-abstraction relationship of the individual, task and development environment. To capture these factors, we developed visualizations for summarizing each collaboration session.

F. Visual Analysis

We divide each session into variable length time segments, where a time segment is an interval in which the work/talk status and abstractions used remains constant. The coupling and abstractions of a segment are represented by zero, one or two shaded rectangles, with rounded or regular corners (Figure 3(a)). If developer 1(2) works during a segment, then a rectangle of regular height is drawn at the high (low) vertical position. The rectangle has regular (rounded) corners, if the developer talks (does not talk) during the segment. If the developers talk without working, then a rectangle of almost zero height is drawn at the middle vertical position. Thus, the height of the shaded area in a segment indicates the number of developers working during that time period. When both developers work, the two rectangles for them (do not) meet seamlessly if they (do not) talk during that time period. Finally, the horizontal position of the left (right) edge of the rectangle(s) indicates the start (end) time of the segment. No rectangle is drawn for the thinking mode.

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8295
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8295

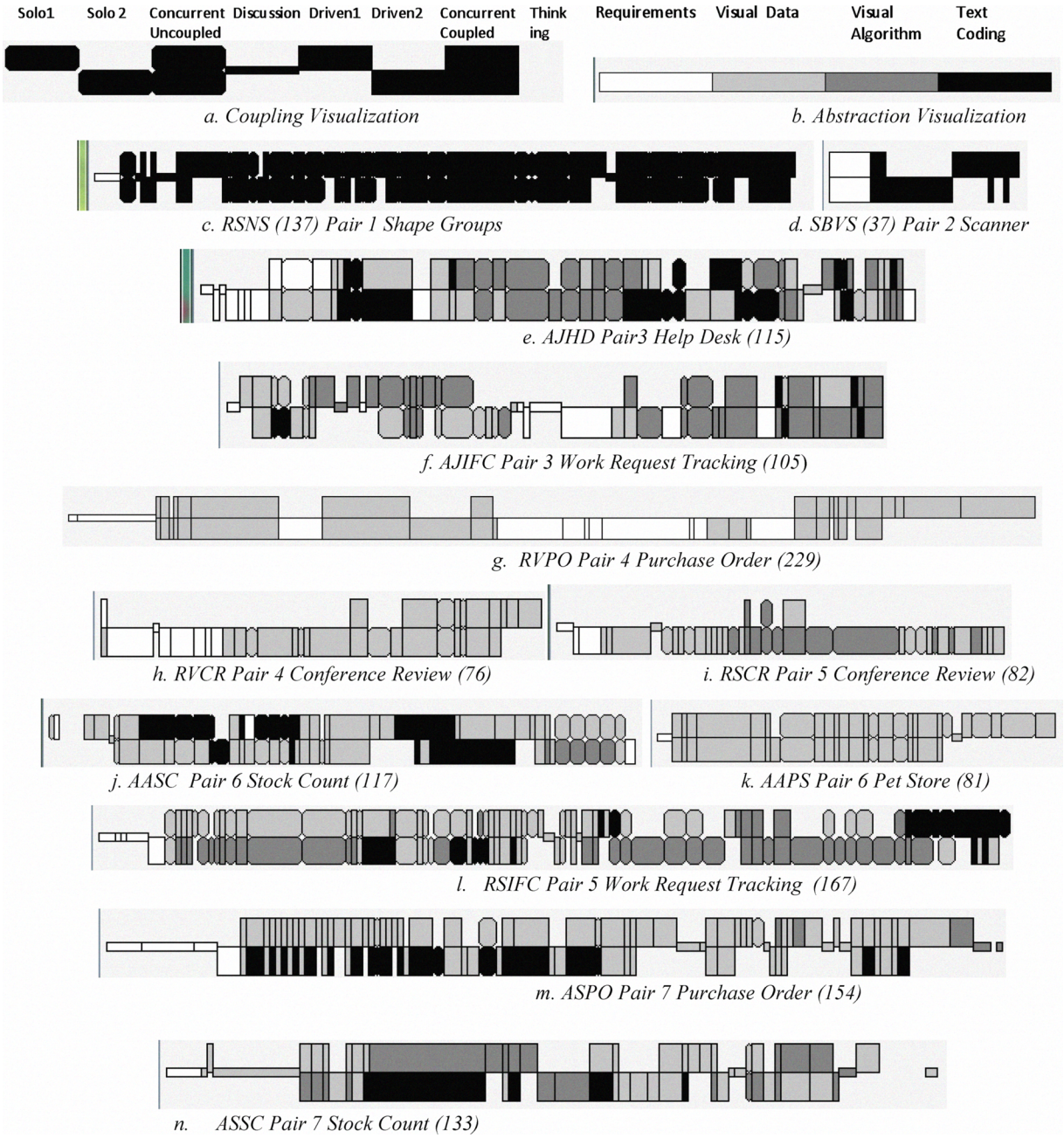

Figure 3. Experiment id, pair id, application name, wall time, and visualization

The shade of a rectangle indicates the level of the abstraction used to perform the associated work - the lighter the shade, the higher the level of the abstraction (Figure 3(b)).

Figure 3 (c-n) visualizes all twelve side-by-side software development sessions we annotated. For each exercise, we show the application and programmer pair. We also show the wall time (half the number of person minutes) taken by the exercise. The reason for the relatively low completion times of the complex Web applications is that InstantApps is an application builder allowing "instant" creation of applications

As one would expect, the nature of the degree to which a mode was used depended on the application. Application dependence is illustrated by the two Microsoft experiments. In the shapes group application of Figure 3(c), most of the time was spent in the two concurrent programming modes, with one programmer modifying the view and controller, and the other changing the model. On the other hand, in the scanning problem of Figure 3(d), which involved changes to

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8295
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8295

only one file, the developers did not use the concurrent programming modes. Based on an examination of the recording, we believe this happened because they were not able to parallelize this task. The architecture used for this experiment required file saves/loads to share edits, and thus did not merge concurrent changes. It is possible that the developers did not even try to parallelize the tasks because of this limitation. The TCS environment, on the other hand, offers concurrency control mechanisms that safely merge edits to different fields of a web page, thereby allowing developers to simultaneously edit the same display. As a result, almost all TCS experiments showed some significant use of the concurrent modes, though some tasks showed less concurrency. In particular, the two TCS pairs who were given the conference review problem did relatively little concurrent programming (Figure 3(h) and (i)). We believe this happened because the task was relatively small and, as a result, more atomic and difficult to parallelize.

Figure 3 also shows individual differences. Different pairs took different amount of times on the same task. For instance, pair 3 and 5 took 105 minutes (Figure 3(f)) and 167 minutes (Figure 3(l)), respectively, on work request tracking. Such differences are consistent with the discussion in Section III.A on comparison perils.

One must carefully analyze the recordings to determine the exact reason for some of the individual differences. However, our abstraction visualization scheme made it easy to spot the reason for one pair - pair 4 never used visual algorithms and text coding on the two tasks given to them (Figure 3(g and h)), even though other pairs who performed the same tasks did do so (Figure 3(i and m)), and in one case, finished the task in much less time. The grading of their work and examination of the recordings showed that pair 4 was an outlier, much slower than the rest, and never completed its tasks.

As mentioned before, our aggregate data showed that all modes were used. Figure 3 shows this was not the case for individual exercises. However, it does show that each session used the unique capabilities of side-by-side development. In particular, every exercise involved the use of the concurrent-coupling and driven modes, and except for the pair 2 scanner exercise (Figure 3(d)), it also involved the use of the concurrent-uncoupled mode, and except for pair 5 conference review exercise (Figure 3(i)), it involved also the use of both driven 1 and 2 modes. This shows that in each exercise, the pair used the unique capability of side-by-side development to (a) support the concurrent-coupling mode, and (b) easily switch between various coupling modes without taking any explicit user-interface action.

G. *Qualitative Analysis*

The visual and quantitative analyses, while relatively succinct, leave several important questions unanswered, which is addressed by the qualitative analysis presented here. It is based on our observations of the live sessions and recordings and answers to questions posed to the subjects.

In any distributed collaboration, performance is an issue. In our collaboration model (Figure 1), response times of primary computers do not include network delays, while those of awareness computers do. We found that though the users had the capability to do so, they never interacted with awareness workstations, perhaps because of the overhead of taking control from their partners. If they had to manipulate an object displayed on that computer, they simply navigated to it on their primary workstation. Network delays are also involved in keeping primary and awareness computers synchronized. The users, connected by a fast LAN, did not find these delays to be an issue. In fact, they said that they felt they were more or less virtually sitting side-by-side modulo occasional problems of gaze awareness wherein it was difficult to know if their partners were looking at their awareness computers.

We know from the quantitative and visual analyses that the unique capabilities of our interaction model (Figure 1) were used, but not the ways in which they were used and how effective they were for different pairs. We present below both some examples to shed light on these questions. Some of the specific scenarios described here were completely unanticipated by us.

Fault tolerant pair programming: One of the unanticipated scenarios actually involved pair programming in which the two computers available to each developer were used for fault tolerance. During the pair programming session of Figure 3(d), the driver's programming environment crashed. The collaboration continued smoothly as the developers changed roles, with the previous navigator now interacting with his programming environment, and the previous driver switching his focus from his primary computer to his awareness computer. This was a rare scenario in that we noticed only one occurrence of it.

Two-display pair programming: Another un-anticipated and more frequent scenario also involved pair programming wherein the driver's primary computer showed the code being developed and the navigator's primary computer the requirements document, During the session of Figure 3(d), the pair was identifying appropriate test cases when the navigator noticed that the requirements document contained test cases they could directly use; thereby significantly reducing task completion time.

Nature of communication: The communication included not only discussion of test cases, as in the example above, but also thinking aloud, rhetorical questions, announcing saves to the file system (in the Microsoft experiments), (solicited and unsolicited) code-inspection and help, discussion of the names, locations, grouping and initial values of form items (in the TCS experiments), clarification of how the programming abstraction/environment worked, algorithm design, asking confirmation for actions such as deletion of a form item, and assignment of tasks to the partner.

Help and Experiment Realism: The fact that several aspects of the collaboration involved giving help to the partner about the programming environment and task justifies our decision to use subjects that were familiar with the programming environment and were qualified to do the

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8295
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8295

problems. Otherwise, there would have been more communication than in a real setting.

Nature of concurrency: Concurrent (coupled/uncoupled) work took several forms. (a) *Concurrent programming*: The developers simultaneously work on their subtasks such as changing the view and model concurrently in the shape groups problem (Figure 3(c)). *Concurrent programming/browsing*: This is an extension of pair programming in which the navigator browses through code and requirements documents, while the driver makes changes to code. For example, in a particular phase of the scanning problem (Figure 3(d)), while the driver was making a change to the code, the navigator browsed through the code to find other locations where a similar change had to be made. *Concurrent browsing*: In this mode, both programmers browse through code. Almost all the concurrent interaction we see in Figure 3 during requirements understanding consisted of such interaction. For example, at the start of the scanning problem (Figure 3(d)), the two programmers concurrently browsed through the code they had to change to try and understand how to change it

Weak typing: As the JavaScript-based InstantApps environment is weakly typed, it allows developers to make references to form items, variables, roles and other identifiers without declaring them. The developers used this feature to work concurrently on dependent subtasks without worrying about compilation/build problems.

Programmer/role: The amount of communication depended on the developer/role. We saw this in a Microsoft pair and at least one TCS pair – most of the communication was initiated by one of the members of the pair. In the case of the Microsoft pair, the communication initiator was the supervisor of his partner – so the asymmetry could be explained by the difference in role and/or experience. In the TCS pair, the initiator was the more extrovert person and also a tester who did not normally do development activities. Thus, the asymmetry could be explained by the difference in experience and/or personalities.

True concurrency vs. fast interleaving in concurrent-coupling: As mentioned earlier, our 1 minute granularity could not always distinguish between interleaving and true concurrency in the concurrent-coupled mode. The granularity would have to be very small to always make this distinction, as we found many cases in which a developer asked for a confirmation, the partner glanced at the awareness computer, approved, and returned to the primary computer, all in less than fifteen seconds. The above is an example of interleaving of work. We also saw developers use peripheral awareness to work and talk concurrently. For example, one of the developers involved in the shape group task (Figure 3(c)), while working on the view, asked his partner working on the model to use relative rather than absolute coordinates.

Conflicts: None of the pairs made conflicting changes in the concurrent programming modes, even though automatic conflict notification was not provided. The reason is that programmers continuously communicated with each other about potentially conflicting changes such as the nature of constructors, order of parameters, names and types of form items, and the names and rights of roles. As the developers did not even try to make conflicting changes, they prevented conflicts even earlier than in continuous conflict resolution[13, 14], which would still be useful for preventing conflicts between different pairs.

Ineffective collaboration: We saw not only positive outcomes of distributed side-by-side development such as conflict prevention and correction of mistakes, but also certain inefficiencies. For example, in the (mostly pair-programming) session of Figure 3(g), one of the developers asked his partner for confirmation of each identifier name, which was always provided by the partner. While such confirmation does increase confidence in one's choices - one of the benefits of pair programming [16] - it also increases the task completion times. We also observed, in the concurrent-coupled mode, a developer repeatedly asking for the attention of her partner, who was resolving a subtle problem with the JavaScript she was writing. This example shows that the quality of advice in the concurrent-coupled mode is probably not as effective as in the driven modes.

IV. Conclusions and Future Work

This work makes several novel contributions. It identifies several of the specific coupling modes that occur in distributed side-by-side development, showing that such development is in fact a union of traditional programming, pair programming, and the concurrent-coupled mode. Furthermore, it gives a meaningful way of partitioning a collaborative session into atomic segments, and presents a way of visualizing the task-coupling relationship in these segments. Finally, it describes results of observing about forty six person hours of such development involving twelve sessions. These results indicates that (a) developers tend to use and switch between all of these modes in realistic exercises; (b) social protocol prevents conflicts in the concurrent modes; and (c) the exact mode used depends on the application, users, abstraction, and environment. To the best of our knowledge, no previous work has made any of these points.

Our work has implications for the design of distributed collaborative technology. It motivates the distributed interaction model of Figure 1 by showing that the unique capabilities of it were used extensively. While the (quantitative, visual and qualitative) analysis we presented to make this point is not bullet proof, to the best of our knowledge, it goes far beyond what has been presented to motivate other synchronous collaboration technologies.

This paper also helps us better understand the practice of distributed collaborative software development by making several observations regarding its use. Task completion times are problematic as a measure of goodness of such development. Given a choice, developers like to mix the collaboration modes of traditional uncoupled development and pair development. When understanding requirements and creating data structures, developers tend to couple their work more than when developing algorithms (visually or textually). In fact, when understanding requirements, they barely use the capabilities of our interaction model, and

This research was funded in part by NSF grants IIS 0312328, IIS 0712794, and IIS 0810861.

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8295
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8295

could probably make do with a simple shared desktop. Some collaborations increase task completion times without improving the productivity or quality of code. If conflicts are a major issue, distributed side-by-side collaboration can help ameliorate the problem.

The extent of our contribution can be better understood by identifying what this paper does not address. It does not claim that our results regarding the practice of distributed side-by-development apply also to the local case. Moreover, it does not show that distributed side-by-side development improves productivity, completion times, or code quality in comparison to other form of collaborative development. In addition, our two display interaction model is not the only approach to support the coupling modes described here. In addition, it would be useful to consider other visualizations that, for instance, provide a fine-grained classification of the work (e.g. browsing vs. editing) and communication (e.g. thinking aloud vs. help) carried out in a session. Furthermore, it would be useful to develop multimedia tools that automatically determine when a developer is talking/working by determining audio-level /screen changes. In addition, it would be useful to determine which display (primary or awareness) each developer views during each time unit to address the gaze awareness problem. It would also be useful to relate the concurrency and communication times in side-by-side programming to those found in traditional software development and radical co-location [22].

In this paper, we have positioned our research with respect to other work in software development. It would be useful to also take a more collaboration-centric view point by comparing it with general research in awareness. In particular, it is important to compare one-display and two-display solutions to awareness. In addition, it would be useful to compare our coupling modes, defined for multi-view distributed collaboration, with those identified in the context of single-view co-located tabletop collaboration [23]. It would also be useful to make our visualizations independent of software development abstractions. Finally, it would be useful to generalize the notion of side-by-side collaboration to more than two users, develop an appropriate visualization of such a collaboration, and determine if it is useful in software development and other activities such as design.

ACKNOWLEDGMENT

Megha Anand coded the TCS recordings and helped create the requirements documents at TCS. Sasa Junuzovic helped design the visualizations.

REFERENCES

[1] Brooks, F., *The Mythical Man-Month.* Datamation, 1974. **20**(12): p. 44-52.

[2] Herbsleb, J.D., A. Mockus, T. A. Finholt, R. E. Grinter . *Distance, dependencies, and delay in a global collaboration.* in *Proc. CSCW*. 2000.

[3] Teasley, S., L. Covi, M. S. Krishnan, J. Olson. *How does radical collocation help a team succeed?* in *Proc. CSCW*. 2000.

[4] Williams, L., et al. *Building Pair Programming Knowledge through a Family of Experiments.* in *IEEE International Symposium on Empirical Software Engineering*. 2003.

[5] Cockburn, A. and L. Williams, *The Costs and Benefits of Pair Programming*. Extreme Programming Examined. 2001: Addison Wesley.

[6] Lui, K.M., K.C.C. Chan, and J. Nosek, *The Effect of Pairs in Program Design Tasks* IEEE Trans. Softw. Eng. , 2008 **34** (2): p. 197-211

[7] Cockburn, A., *Crystal Clear. A Human-Powered Methodology for Small Teams*. 2005: Addison-Wesley.

[8] Nawrocki, J.R., et al., *Pair Programming vs. Side-by-Side Programming*, in *Software Process Improvement*. 2005, Springer Berlin / Heidelberg. p. 28-38.

[9] Dewan, P., P. Agarwal, G. Shroff, R. Hegde. *Distributed Side-by-Side Programming*. in *2009 ICSE CHASE Workshop* . IEEE.

[10] Shroff, G., P. Agarwal, and P. Devanbu. *Instant Multi-tier Applications without Tears,*. in *2nd India Software Engineering Conference*. 2009. Pune, India.

[11] Redmiles, D., et al., *Continuous Coordination: A New Paradigm to Support Globally Distributed Software Development Projects.* Wirtschaftsinformatik, 2007. **49 (Special Issue)**: p. 28-38.

[12] Cheng, L.-T., et al. *Jazzing up Eclipse with collaborative tools.* in *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*. 2003.

[13] Sarma, A., B. G, and A.v.d. Hoek. *Towards Supporting Awareness of Indirect Conflicts across Software Configuration Management Workspaces.* in *Twenty-second IEEE/ACM ASE* 2007. Atlanta, Georgia.

[14] Dewan, P. and R. Hegde. *Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development.* in *ECSCW*. 2007.

[15] Dewan, P., R. Choudhary, and H. Shen, *An Editing-based Characterization of the Design Space of Collaborative Applications.* Journal of Organizational Computing, 1994. **4**(3): p. 219-240.

[16] Nosek, J.T., *The Case for Collaborative Progamming.* CACM, 1998. **41**(3): p. 105-108.

[17] Nawrocki, J. and A. Wojciehowski. *Experimental Evaluation of Pair Programming.* in *European Software Control and Metrics*. 2001. London.

[18] Ratcliffe, T.L. and A. Robertson. *Code Warriors and Code-a-Phobes: A study in attitude and pair programming.* in *SIGCSE*. 2003.

[19] Padberg, F. and M. Muller, *Analyzing the Cost and Benefit of Pair Programming*, in *Proceedings of the 9th International Symposium on Software Metrics*. 2003, IEEE Computer Society. p. 166.

[20] Baheti, P., E.F. Gehringer, and P.D. Stotts, *Exploring the Efficacy of Distributed Pair Programming* in *XP/Agile Universe 2002* p. 208-220

[21] Wolf, T., T. Nguyen, and D. Damian, *Does distance still matter?* Softw. Process 2008 **13** (6): p. 493-510

[22] Begel, A. and B. Simon. *Novice software developers, all over again.* in *International Computing Education Research Workshop*. 2008.

[23] Tang, A., et al. *Collaborative Coupling over Tabletop Displays.* in *CHI*. 2006.

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8295
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8295