# Topology Awareness for Distributed Version Control Systems

Cristiano M. Cesario and Leonardo G. P. Murta
Instituto de Computação, Universidade Federal Fluminense (UFF)
Niteroi, RJ, Brazil
{ccesario, leomurta}@ic.uff.br

## ABSTRACT

Software development using distributed version control systems has become more frequent recently. Such systems bring more flexibility, but also greater complexity to manage and monitor the multiple existing repositories as well as their myriad of branches. In this paper, we propose DyeVC, an approach to assist developers and repository administrators in identifying dependencies among clones of distributed repositories. It allows understanding what is going on around one's clone and depicting the relationship between existing clones. DyeVC was evaluated over open source projects, showing how they could benefit from having such kind of tool in place. We also ran an observational study and a benchmark over DyeVC, and the results were promising: it was considered easy to use and fast for most repository history exploration operations, while providing the expected answers.

## Categories and Subject Descriptors

• *Software and its engineering~Software version control*

## Keywords

Topology Awareness, Distributed Version Control.

## 1. INTRODUCTION

Version Control Systems (VCS) date back to the 70s, when SCCS emerged [23]. Their primary purpose is to keep software development under control [13]. Along these almost 40 years, VCSs have evolved from a centralized repository with local access (e.g., SCCS and RCS [28]) to a client-server architecture (e.g., CVS [4] and Subversion [8]). More recently, distributed VCSs (DVCS) arose (e.g., Git [6] and Mercurial [21]) allowing clones of the entire repository in different locations. According to a survey conducted by the Eclipse community [11], Git and Github combined usage increased from 6.8% to 42.9% between 2010 and 2014 (a growth greater than 500%). During this same period, Subversion and CVS combined usage decreased from 71% to 34.4%. This clearly shows momentum and a strong tendency in the adoption of DVCSs in the open source community.

Besides these changes from local to client-server and then to a distributed architecture, the concurrency control policy adopted by VCSs also changed from lock-based (pessimistic) to branch-based (optimistic). According to Walrad and Strom [30], creating branches in VCSs is essential to software development because it enables concurrent development, allowing the maintenance of different versions of a system, the customization to different platforms/customers, among other features. DVCSs include better support for working with branches [20], turning the branch creation into a recurring pattern, no matter if this creation is explicitly done by executing a "branch" command or implicitly when a repository is cloned.

However, distributed software development, especially from the geographical perspective [18], brings a set of risk factors, and Configuration Management (CM) is affected by them. The increasing growth of development teams and their distribution along distant locations, together with the proliferation of branches, introduce additional complexity for perceiving actions performed in parallel by different developers. According to Perry et al. [22], concurrent development increases the number of defects in software. Besides, Silva et al. [25] say that branches are frequently used for promoting isolation among developers, postponing the perception of conflicts that result from changes made by co-workers. These conflicts are noticed only after pulling changes in the context of DVCSs. Moreover, Brun et al. [3] show that, even using modern DVCSs, conflicts during merges are frequent, persistent, and appear not only as overlapping textual edits (i.e., physical conflicts) but also as subsequent build (i.e., syntactic conflicts) and test failures (i.e., semantic conflicts).

By enabling repository clones, DVCSs expand the branching possibilities discussed by Appleton et al. [1], allowing several repositories to coexist with fragments of the project history. This may lead to complex topologies where changes can be sent to or received from any clone. This scenario generates traffic similar to that of peer-to-peer applications. In practice, projects impose some restrictions over this topology freedom. However, it can be still much more complex than the traditional client-server topology found in centralized VCS.

With this diversity of topologies, managing the evolution of a complex system becomes a tough task, making it difficult to find answers to the following questions: (Q1) Which clones were created from a repository? (Q2) What are the communication paths among different clones? (Q3) Which changes are under work in parallel (in different clones or different branches) and which of them are available to be incorporated into others' clones? Most of the existing works, such as Palantir [24], FASTDash [2], Lighthouse [25], CollabVS [9], Safe-Commit [31], Crystal [3], and WeCode [17], deal with question Q3, giving to the developers awareness of concurrent changes. However, they do not provide an overview of the topology of repositories, indicating which commits belong to which clones. This overview is essential to understand the distributed evolution of the project.

In order to answer the aforementioned questions, we propose

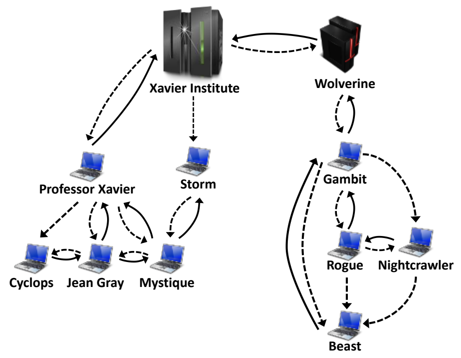


**Figure 1. Development scenario involving some developers**

DyeVC[1], a novel monitoring and visualization approach for DVCS that gathers information about different repositories and presents them visually to the user. DyeVC allows developers to perceive how their repository evolved over time and how this evolution compares to the evolution of other repositories in the project. DyeVC's main goal is two-fold: increasing the developers' knowledge of what is going on around their repository and the repositories of their teammates, and enabling repository administrators to visualize the relationship between existing clones.

This paper expands the concepts presented in a previous workshop paper [5] by including a more thorough discussion about our approach, such as how we depict the dependencies among the repositories in a project. It also includes the evaluation of DyeVC over open source projects and the results of an observational study and a benchmark over DyeVC. This paper is organized as follows: Section 3 presents the DyeVC approach. Section 4 presents the technologies used in our prototype implementation. Section 5 describes the evaluation of DyeVC. Section 6 discusses related work and Section 7 concludes the paper and presents some future work.

## 2. MOTIVATIONAL EXAMPLE

Figure 1 shows a scenario with some developers, each one owning a clone of the repository originally created at Xavier Institute. Xavier Institute acts like a central repository, where code developed by all teams is integrated, tested, and released to production. There is a team working at Xavier Institute, led by Professor Xavier, and a remote developer (Storm) that periodically receives updates from the Institute. Outside the Institute, Wolverine leads a remote team located in a different site, which is constantly synchronized with the Institute. Solid lines in Figure 1 indicate data being pushed, whereas dotted lines indicate data being pulled. Thus, for example, Rogue can both pull updates from Gambit and push updates to him, and Beast can pull updates from Rogue, but cannot push updates to her.

Each one of the developers has a complete copy of the repository. Luckily, this scenario has a CM Plan in action, otherwise each one would be able to send and receive updates to and from any other, leading to a total of $n \times (n-1)$ different possibilities of communication (where n is the number of developers in the topology). In practice, however, this limit is not reached: while interaction amongst some developers is frequent, it may happen that others have no idea about the existence of some coworkers. It occurs with Mystique and Nightcrawler, for example, where there is no direct communication.

As an example, from a developer's point of view, like Beast, how can he know at a given moment if there are commits in Rogue, in Gambit, or in Nightcrawler clones that were not pulled yet? Alternatively, would be the case that there are local commits pending to be pushed to Gambit? Beast could certainly periodically pull changes from his peers, checking if there were updates available, but this would be a manual procedure, prone to be forgotten. It would be more practical if Beast could have an up to date knowledge of his peers, warning him about any local or remote updates that had not been synchronized yet. On the other hand, from an administrator's point of view, how can she know which are the existing clones of a project and how they relate among each other? How can she know if there are pending commits to be sent from a staging repository to a production one?

## 3. DYEVC APPROACH

Aiming at supporting both developers and repository administrators in understanding the interaction of repository clones, the main features of DyeVC include: (1) a mechanism to gather information from a set of clones and (2) a set of extensible views with different levels of detail, which let DyeVC users visualize this information. We detail in the following sub-section how DyeVC gathers information from DVCSs. Next, we discuss how this information is presented using different levels of detail. Finally, we show what happens behind the scenes, discussing the algorithm involved in the data synchronization process.

### 3.1 Information Gathering

DyeVC continuously gathers information from interrelated clones, starting from clones registered by the user. For each registered clone *rep*, DyeVC transparently creates a local clone *rep'* in the user's home folder to fetch data from all of the peers that *rep* communicates with. Data is gathered by DyeVC instances running at each user machine and is stored in a central document database. In this way, information from one DyeVC instance is made available to every other instance in the topology.

DyeVC gathers information from registered clones in the user's machine and also from their peers, which are clones that communicate with them. Since there is a communication path between a registered clone and its peers (either to push or pull data), we are able to analyze the commits that exist in these peers. This allows us to present a broader topology visualization that contains not only registered clones, but also those that have a push or pull relationship with them. DyeVC finds out related clones by looking at the remote repositories registered in the DVCS configuration. More details on how data is gathered are explained in section 3.3.

Data stored at the central database follows the metamodel presented in Figure 2. A *Project* groups repository clones of the same system. Clones are stored as *RepositoryInfo* and are identified by an id and a meaningful clone name provided by the user. A *RepositoryInfo* has a list of clones to which it pushes data and a list of clones from which it pulls data. These lists are represented respectively by the self-associations *pushesTo* and *pullsFrom*. Finally, a *RepositoryInfo* stores the hostname where it resides (e.g., a server name or *localhost*) and its path (be it an operating system path or a URL).

Branches are part of a *RepositoryInfo*. A *Branch* has a name and a boolean attribute *isTracked*, which is true if the branch tracks a remote branch. A *RepositoryInfo* may have one or many branches

[1] Dye is commonly used in cells to observe the cell division process. As an analogy, DyeVC allows developers to observe how a Version Control repository evolved over time.

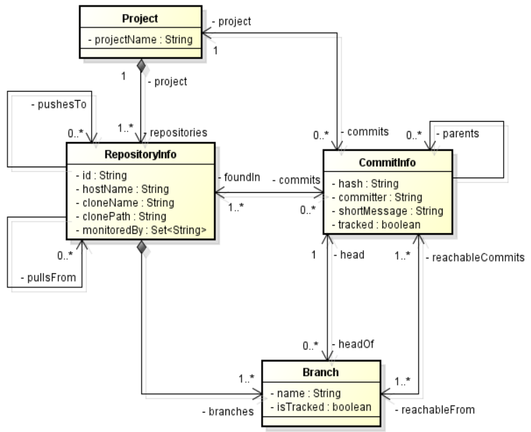


**Figure 2. Metamodel used to store DyeVC data**

(it must have at least one branch, which is the main one). A *Branch* has two associations with *CommitInfo*: through the first association, a *Branch* knows which commit is its head and, conversely, a commit knows which branches point to it as a head. The second association represents which commits are reachable from a given branch and, conversely, the branches from which the commit is reachable.

The finer grain of information is the *CommitInfo*, which represents each commit in the topology. A commit is identified by a hash code and it refers to its parents (except for the first commit in the repository, which does not have any parent). As each commit may not exist in all clones of the topology, we store the list of clones where each commit can be found (*foundIn* association). We also store the committer, the commit message, and whether the commits belong to tracked branches or to non-tracked branches.

## 3.2 Information Visualization

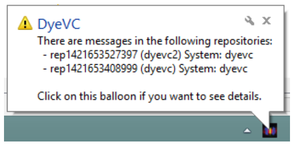
DyeVC presents information in four different levels of detail: Level 1 shows high-level notifications about registered repositories; Level 2 shows the whole topology of a given project. Level 3 zooms into the branches of the repository, showing the status of each tracked branch. Lastly, Level 4 zooms into the commits of the repository, showing a visual log with information about each commit. The following sections discuss these levels.

### *3.2.1 Level 1: Notifications*

In Level 1, our approach periodically monitors registered repositories and presents notifications whenever a change is detected in any known peer. The period between subsequent runs is configurable, and notifications are presented in the system notification area, in a non-obtrusive way. Figure 3 shows an example of this kind of notification, where DyeVC detected changes in two different repositories. The notification shows the repository id, the clone name, and the project (system) name. Clicking on the balloon opens DyeVC main screen.

### *3.2.2 Level 2: Topology*

Aiming at helping answering questions Q1 and Q2, we present a topology view showing all repositories for a given project (Figure 4), where each node represents a known clone. A blue computer represents the current user clone and black computers represent other clones where DyeVC is running. Servers represent central repositories, that do not pull from nor push to any other clone, or clones where DyeVC is not running. Both kinds of nodes use the same representation because, once DyeVC is not running at a given clone, we cannot infer the *pushesTo* and *pullsFrom* lists, which will thus be empty as in a server.


DyeVC
There are messages in the following repositories:
- rep1421653527397 (dyevc2) System: dyevc
- rep1421653408999 (dyevc) System: dyevc

Click on this balloon if you want to see details.

**Figure 3. DyeVC showing notifications in notification area**

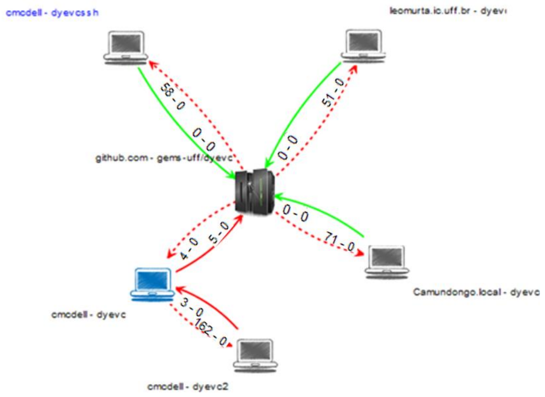


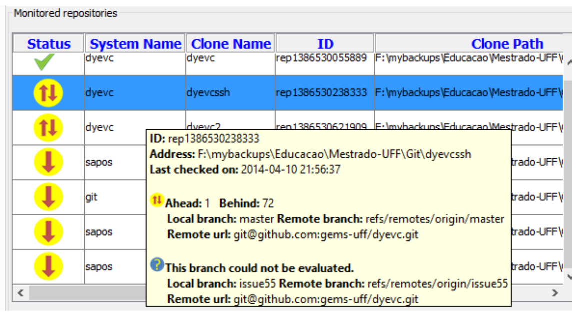
**Figure 4. Topology view for a given project**

Each edge in the graph represents a relationship between two repositories. Continuous edges mean that the source clone pushes to the destination clone, whereas dashed edges mean that the destination clone pulls from the source clone. The edge labels show two numbers separated by a dash. The first and second numbers represent how many commits in tracked and non-tracked branches of the source clone are missing in the destination clone, respectivelly. The edge colors are used to represent the synchronization status: green edges mean that both clones are synchronized (i.e., the destination clone has all commits in the source clone), whereas red edges mean that the pair is not synchronized and indicates the direction that is missing commits. For example, it is possible to observe in Figure 4 that the current user clone (blue computer) is hosted at *cmcdell* and is named dyevc. This clone pulls from *gems-uff/dyevc*, which is located at github.com, and there are four tracked commits ready to be pulled (i.e., commits that exist in the remote repository and do not exist locally). It also pushes to the same peer, having five tracked commits ready to be pushed.

### *3.2.3 Level3: Tracked branches*

Aiming at answering question Q3, DyeVC's main screen (see Figure 5) shows Level 3 information, allowing one to depict the status of each tracked branch of registered repositories regarding their peers. This information is complemented with that of Level 4, shown in the next section.

The status evaluation considers the existing commits in each repository individually. Due to the nature of DVCS, old data is almost never deleted and commits are cumulative. Thus, if commit *N* is created over commit *N – 1*, the existence of commit *N* in a given repository implies that commit *N – 1* also exists in the repository. In this way, by using set theory it is possible to subtract the set of commits in the local repository from the set of commits in its peers, resulting in the set of commits not pulled yet.


Monitored repositories

| Status | System Name | Clone Name | ID | Clone Path |
|---|---|---|---|---|
| ✓ | dyevc | dyevc | rep1386530055889 | F:\mybackups\Educacao\Mestrado-UFF\ |
| ⇅ | dyevc | dyevcssh | rep1386530238333 | F:\mybackups\Educacao\Mestrado-UFF\ |
| ⇅ | dyevc | dyevc2 | rep1386530621909 | F:\mybackups\Educacao\Mestrado-UFF\ |
| ↓ | sapos | | | trado-UFF\ |
| ↓ | git | | | trado-UFF\ |
| ↓ | sapos | | | trado-UFF\ |
| ↓ | sapos | | | trado-UFF\ |

**ID:** rep1386530238333
**Address:** F:\mybackups\Educacao\Mestrado-UFF\Git\dyevcssh
**Last checked on:** 2014-04-10 21:56:37

⇅ **Ahead:** 1 **Behind:** 72
**Local branch:** master **Remote branch:** refs/remotes/origin/master
**Remote url:** git@github.com:gems-uff/dyevc.git

? **This branch could not be evaluated.**
**Local branch:** issue55 **Remote branch:** refs/remotes/origin/issue55
**Remote url:** git@github.com:gems-uff/dyevc.git

**Figure 5. DyeVC main screen**

In this case, local repository will be *behind* its peers (arrow down in Figure 5). Conversely, subtracting the sets in the inverse order will result in the set of commits not pushed yet, meaning that local repository is ahead its peers (arrow up). When both sets are empty, local repository is synchronized (green checkmark in Figure 5) and when both sets have elements, it is both ahead and behind its peer (arrow up and down in Figure 5).

Let us assume that each commit is represented by an integer number to illustrate how our approach works. At a giving moment, the local repositories of each developer have the commits shown in Table 1. Considering the synchronization paths presented in the right-hand-side of Figure 1, the perception of each developer regarding to their known peers is shown in Table 2. Notice that the perceptions are not symmetric. For instance, as Gambit does not pull updates from Nightcrawler, there is no sense in giving him information regarding Nightcrawler. Furthermore, it is uncommon to have a scenario where pushes are performed from a developer to another (such as the one between Beast and Gambit). Generally, what happens is that a developer pulls from another (for example, between Gambit and Nightcrawler). This avoids inadvertent inclusion of commits inside others' clones. Although infrequent, this scenario helps in understanding the need to have awareness about who are the peers in a project and what are their interdependencies.

### *3.2.4 Level 4: Commits*

Level 4 complements information of Level 3 in order to provide an answer to Question Q3. Differently from the usual repository version graph, it presents a combined version graph of the whole topology (Figure 6). Each vertex in the graph represents either a known commit in the topology (which is named after its hash's five initial characters, such as the node labeled *2e10a*) or a collapsed node representing several commits (which is named after the number of contained nodes, such as the white node containing 118 commits and the green node containing 24 commits). Collapsing is done manually and is a way of providing a better understanding over huge amounts of data. Thicker borders denote that the commit is a branch's head (e.g., commit *ea6a4*). Commits are drawn according to their precedence order. Thus, if a commit *N* is created over a commit *N – 1*, then commit *N* will be located to the right of commit *N – 1*. For each commit, DyeVC presents the information described in Figure 2 (gathered from the central database), along with information that is read in real time from the repository metadata, such as branches that point to that commit and affected files (added, edited, and deleted).

This visualization contains all commits of all clones in an integrated graph. Each commit is painted according to its existence in the local repository and in the peers' repositories. Ordinary commits that exist locally and in all peers are painted in white. Green commits are ready to be pushed, as they exist locally but do not exist in peers of the push list. Yellow commits need attention because they exist in at least one peer in the pull list, but do not exist locally, meaning that they may be pulled. Red commits do not exist locally and are not available to be pulled, as they exist only in clones that are not peers. Finally, gray commits belong to non-tracked branches, so they can neither be pushed nor pulled. Heads in these branches are not identified with thicker borders.

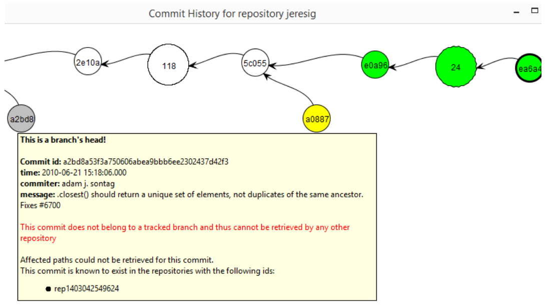
This visualization can easily have thousands of nodes, one for each commit in the topology. Nevertheless, despite the high amount of nodes, users are generally interested in the most recent commits. As we show the commits following a chronological order, from left to right, most recent commits will be at the right part of the visualization. DyeVC positions the graph so that these commits are shown when opening the visualization.

## 3.3 Behind the Scenes

Algorithm 1 shows how DyeVC updates its database with commits in the topology. The process is based on Set Theory and is executed for each repository *rep* monitored by DyeVC. The algorithm receives as input the repository being monitored (*rep*), the set of existing commits in the database (*db.commits*), the set of existing commits in *rep* at the previous monitoring cycle (*previousSnapshot*), and the set of existing commits in *rep* at the current monitoring cycle (*currentSnapshot*).

First of all, we subtract *currentSnapshot* from *previousSnapshot* to find *commitsToDelete*, that contains commits that were deleted since the previous monitoring cycle (line 2) and we delete them from the database, in order to cover the rare situations where a commit is deleted (line 3). Conversely, we subtract *previousSnapshot* from *currentSnapshot* to find *newCommits*, which contains commits that are new in *rep* since the previous monitoring cycle (line 5).

Next, we find out which commits in *newCommits* need to be



**This is a branch's head!**

**Commit id:** a2bd8a53f3a750606abea9bbb6ee2302437d42f3
**time:** 2010-06-21 15:18:06.000
**commiter:** adam j. sontag
**message:** .closest() should return a unique set of elements, not duplicates of the same ancestor. Fixes #6700

This commit does not belong to a tracked branch and thus cannot be retrieved by any other repository

Affected paths could not be retrieved for this commit.
This commit is known to exist in the repositories with the following ids:

- rep1403042549624

**Figure 6. Collapsed commit history**

**Table 1. Existing Commits in Each Repository**

| *Repository* | *Wolverine* | *Gambit* | *Rogue* | *Nightcrawler* | *Beast* |
|---|---|---|---|---|---|
| Commits | 10; 11 | 10; 11 | 10; 12 | 10; 11; 13 | 10 |

**Table 2. Status of Each Repository Based on Known Remote Repositories**

| *Repository* | *Wolverine* | *Gambit* | *Rogue* | *Nightcrawler* | *Beast* |
|---|---|---|---|---|---|
| Wolverine | - | - | - | - | - |
| Gambit | ✓ | - | - | - | - |
| Rogue | - | ⇅ | - | - | - |
| Nightcrawler | - | ✓ | ⇅ | - | - |
| Beast | - | ↓ | ↓ | ↓ | - |

inserted into the database, by subtracting the existing commits in the database (db.*commits*) from *newCommits* (line 6). This step is necessary because some of the new commits might have already been inserted into the database by another instance of DyeVC. Commits that might need to be updated are represented by *commitsToUpdate* (line 7). They consist of those commits that exist in the database, but were not found in at least one of the repositories related to *rep* on the last monitoring cycle.

Commits to be inserted or updated must be verified to check where they exist, thus updating the *c.foundIn* attribute. This verification is done using the procedure *updateFoundIn* (lines 23-45), which is called in lines 10 and 14. This procedure finds out where each commit *c* exists based on its local existence or in any repository in the push or pull lists. This procedure verifies if *rep* is ahead of any repository in its push list regarding *c* (line 24), i.e., if *c* exists and if there is at least one repository that *rep* pushes to that does not contain *c*. Likewise, it verifies if *rep* is behind of any repository in its pull list regarding *c* (line 25), i.e., if *c* does not exist locally and if there is at least one repository that *rep* pulls from that contains *c*. If *rep* is behind, then all repositories in *rep's* pull list that contain *c* are added to *c.foundIn* (lines 27-29). If *rep* is ahead, then *rep* and all repositories in *rep's* push list that contain *c* are added to *c.foundIn* list (lines 30-32). It may happen that *rep* is neither ahead nor behind any repository (line 33). In such case, one of the following three scenarios may happen: In scenario 1, *c* does not exist in the current snapshot (line 34), meaning that it also does not exist in any of the related repositories, thus we remove *rep* and all its related repositories from *c.foundIn* (line 35). For scenarios 2 and 3, we first depict if *c* is reachable from a tracked branch, i.e., if at least one of *rep.branches* is tracked and has *c* as one of its commits (line 37). In scenario 2, *c* is in a tracked branch, meaning that it also exists in all related repositories (remember that *rep* is neither ahead nor behind their partners), thus we include *rep* and all its related repositories in *c.foundIn* (line 39). Finally, in scenario 3, *c* is not in a tracked branch, meaning that it exists only in *rep*, thus we include only *rep* in *c.foundIn* (line 41).

After updating where each commit is found, commits in *commitsToInsert* are inserted into the database (line 17) and commits in *commitsToUpdate* are updated in the database (line 18). Finally, it may happen that some commits end up with an empty *foundIn* attribute, meaning that they do not exist anywhere in the topology (line 19). These so-called *orphanedCommits* are then removed from the database (line 20) and the algorithm ends.

## 4. IMPLEMENTATION

We implemented our approach as a Java application launched via Java Web Start Technology. It currently monitors Git repositories, as it is the most used DVCS nowadays [11]. The source code and the link to download the tool via Java Web Start can be found at https://github.com/gems-uff/dyevc. The tool gathers information from repositories using JGit library[2], which allows using our approach without having a Git client installed.

Gathered information is stored in a central document database running MongoDB. We hosted our database on a free MongoDB instance provided by MongoLab. We did not use MongoDB proprietary API, which would demand opening specific ports to connect to MongoDB. Instead, we opted to use MongoLab's RESTful (*Representational State Transfer*) API. RESTful APIs [14] have the advantage of being available using standard HTTP and HTTPS protocols. In this way, our approach can be used in environments protected with firewalls without major problems. In order to use this RESTful API, we implemented a *MongoLabProvider*, which translates the application methods into RESTful commands and vice-versa. It also serializes/deserializes the application objects to/from JSON (*JavaScript Object Notation*) repre-

**Algorithm 1: Updating commits in the topology**
**input**: a *RepositoryInfo rep* representing the repository being analyzed and three sets of *CommitInfo db.commits*, *previousSnapshot*, and *currentSnapshot*.

```
1:  begin
2:  $commitsToDelete \leftarrow previousSnapshot \setminus currentSnapshot$
3:  delete $commitsToDelete$ from $database$
4:
5:  $newCommits \leftarrow currentSnapshot \setminus previousSnapshot$
6:  $commitsToInsert \leftarrow newCommits \setminus db.commits$
7:  $commitsToUpdate \leftarrow \{c \mid c \in db.commits \wedge$
        $((rep.pullsFrom \cup rep.pushesTo) \setminus c.foundIn) \neq \emptyset\}$
8:
9:  for each $c \in commitsToInsert$ do
10:     UPDATEFOUNDIN($c$, $rep$, $currentSnapshot$)
11: end for
12:
13: for each $c \in commitsToUpdate$ do
14:     UPDATEFOUNDIN($c$, $rep$, $currentSnapshot$)
15: end for
16:
17: insert commitsToInsert into $database$
18: update commitsToUpdate in $database$
19: $orphanedCommits \leftarrow \{c \mid c \in db.commits \wedge c.foundIn =$
        $\emptyset\}$
20: delete orphanedCommits from $database$
21: end
22:
23: procedure UPDATEFOUNDIN(c: CommitInfo, rep: RepositoryInfo,
        currentSnapshot: Set of CommitInfo)
24:     $isAhead \leftarrow c \in currentSnapshot \wedge$
            $\exists r\,(r \in rep.pushesTo \wedge c \notin r.commits)$
25:     $isBehind \leftarrow c \notin currentSnapshot \wedge$
            $\exists r\,(r \in rep.pullsFrom \wedge c \in r.commits)$
26:
27:     if $isBehind$ then
28:         $c.foundIn \leftarrow c.foundIn \cup \{r \mid r \in rep.pullsFrom \wedge$
        $c \in r.commits\}$
29:     end if
30:     if $isAhead$ then
31:         $c.foundIn \leftarrow c.foundIn \cup rep \cup$
                $\{r \mid r \in rep.pushesTo \wedge c \in r.commits\}$
32:     end if
33:     if $(\neg\,(isBehind) \wedge \neg\,(isAhead))$ then
34:         if $c \notin currentSnapshot$ then
35:             $c.foundIn \leftarrow ((c.foundIn \setminus rep) \setminus rep.pushesTo) \setminus$
                    $rep.pullsFrom$
36:         else
37:             $isTracked \leftarrow \exists b(b \in rep.branches \wedge$
                    $c \in b.reachableCommits)$
38:             if $isTracked$ then
39:                 $c.foundIn \leftarrow c.foundIn \cup rep \cup$
                        $rep.pushesTo \cup rep.pullsFrom$
40:             else
41:                 $c.foundIn \leftarrow rep$
42:             end if
43:         end if
44:     end if
45: end procedure
```

[2] http://www.eclipse.org/jgit/

sentations to be used through the RESTful commands.

We present the gathered information as a series of graphs by using the JUNG (*Java Universal Network/Graph*) library[3], from which DyeVC inherits the ability to extend existing layouts and filters. All graphs present similar behavior, allowing the window to be zoomed in or out, whether the user wants to see details of a particular area or an overview of the entire graph. By changing the window mode from *transforming* to *picking*, it is possible to select a group of nodes and collapse them into one node, or simply drag them into new positions to have a better understanding of parts with too many crossing lines.

## 5. EVALUATION

In order to evaluate our approach we performed two experiments and an observational study. First, we conducted a *post-hoc* analysis over the JQuery project[4], an open-source project, aiming at checking if DyeVC can help answering questions Q1-Q3. Next, we conducted an observational study involving four participants that used DyeVC. This study also used the JQuery project. Finally, we run DyeVC over some open-source projects of different sizes and from different sources, aiming at evaluating the scalability of our approach.

### 5.1 Post-hoc Study

We conducted a *post-hoc* analysis using a real open source project to demonstrate that our approach can help answering questions Q1-Q3. The selected project, JQuery, began in 2006 and had 6,222 commits by the time of the evaluation. We reconstructed the repository history, simulating the actions that occurred in the past. We do not replicate the repository history here, due to its size, but it is publicly available at Github. Automatically generated comments helped us to depict specific flows. For example, the comment "*Merge branch 'master' of https://github.com/scottjehl/jquery into scottjehl-master*" tells us that there was a user named "*scottjehl*" and that the merge operation was done at a branch called "*scottjehl-master*". Although one might perform a merge manually and insert a different text in the comment, this did not compromise our analysis because we had a focus on depicting some of the merge situations, and not all of them.

Due to the operating mode of Git, some details are missing, but these details do not compromise our analysis. The first one is the moment when a clone arises or deceases. This information does not exist anywhere in the repository. We inferred the creation of clones by looking at the commit messages (a commit by developer X led to the creation of a clone named X). Clones created at a given time stayed alive for the rest of the analysis.

The second missing detail is that, although we had the commit dates and times in the repository history, these dates and times were not guaranteed to be correct. This occurs because DVCSs do not have a central clock. Each commit is registered with the local time at the machine where the clone is located, which could lead to commits in the history with a predecessor in the future, depending on when and where each commit was performed. This missing detail is not important, because the order of commits is not depicted using their times, but using the pointers that Git maintains from a commit to its parents, as discussed in section 3.1. We can use these dates, but not as an authoritative information.

We chose a moment in time when three developers were involved, performing commits and merging changes in the repository. We created three clones for these developers, named after their user names: *jeresig*, *adam*, and *aakosh*. Figure 7 shows the topology view on Sep 24 2010, when *aakosh* had 121 commits pending to be pushed to the central repository (hereafter called *central-repo*). Figure 8 shows part of *aakosh's* commit history and how DyeVC represents commits pending to be pushed (green nodes).

Later on, *aakoch* pushed his commits to *central-repo*. In the meantime, both *adam* and *jeresig* committed some changes. Before they pushed their work to *central-repo*, *adam's* last commit was on Jun 21 2010 and *jeresig's* on Sep 27 2010. At this moment, we registered them to be monitored by DyeVC. Figure 9 shows the topology view after this registration on Sep 27 2010. Here, we can see that *aakoch* was synchronized with *central-repo*, whereas *adam* and *jeresig* had pending actions. At this point, we can revisit questions Q1 and Q2:

Q1: *Which clones were created from a repository*? DyeVC's topology view (Figure 9) shows all the clones where it is running, and also discovers other clones connected to them, even if it is not running there.

Q2: *What are the communication paths among different clones?* DyeVC's topology view (Figure 9) shows the dependencies between the peers in the topology, as well as the number of commits ahead or behind in each of these clones.

*Adam* had 121 commits to pull from *central-repo*, what is corroborated by the details of his tracked branches (master branch in Figure 10). He also had a non-tracked commit pending to be pushed. Non-tracked commits are not shown in the tracked branches view, but we can see them in gray in the commit history views. Figure 6 shows the collapsed commit history for *jeresig*, where we can see adam's non tracked commit with hash *a2bd8*.

The repository history leads us to think that *jeresig* is a core developer of this project, because he performed most of the merges to the master branch. Looking at Figure 9, we see that he had 26 commits pending to be pushed to *central-repo*. These 26 commits can be seen at *aakoch's* commit history (Figure 11) as red commits, since they could not be pulled by *aakoch* until *jeresig* has pushed them to *central-repo*. There was also a commit in central-repo pending to be pulled by *jeresig*. If we look back at Figure 6 we see that the only yellow commit is *a0887*, made by *aakoch*. This tells us that *jeresig* pulled changes from *central-repo* just

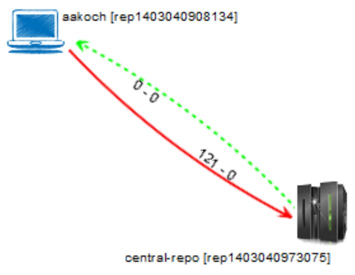


**Figure 7. First monitored repository in Topology view (Sep 24 2010)**

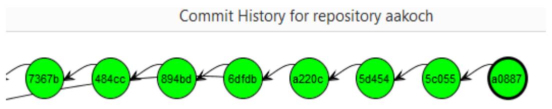


**Figure 8. aakoch's commit history showing commits pending to be pushed**

[3] http://jung.sourceforge.net/

[4] https://github.com/jquery/jquery

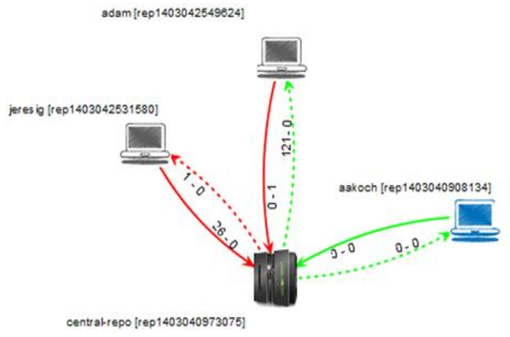

adam [rep1403042549624]
jeresig [rep1403042531580]
aakoch [rep1403040908134]
central-repo [rep1403040973075]
121 - 0
0 - 1
1 - 0
26 - 0
0 - 0
0 - 0

**Figure 9. Three monitored repositories in Topology view (Sep 27 2010)**

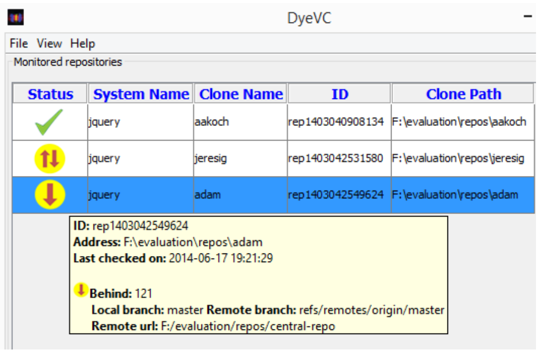
before *aakoch* pushed commit *a0887*. If we look at Figure 12, we see that all the pending commits (those that were pending to be pushed and pulled) are related to the same branch (master). This tells us that, if *jeresig* wanted to push these commits to *central-repo*, he would have to perform a pull operation before. This analysis helps us revisit and answer Q3:

Q3: *Which changes are under work in parallel (in different clones or different branches) and which of them are available to be incorporated into others' clones?* New commits in tracked branches of peers can be easily found by looking at Level 3 information (tracked branches, shown in Figure 10 and Figure 12). This view shows to which branch these commits are related and how many new commits exist. If we want to look at each commit individually, we can look at Level 4 information (commit history, shown in Figure 8 and Figure 11) and notice the yellow nodes. Additionally, Level 4 information can be used to find new commits in repositories that are not peers (red nodes), or new commits in non-tracked branches (gray nodes).

## 5.2 Observational Study

We conducted an observational study over the same project used in the post hoc analysis (JQuery) to assess the capability of the visualizations provided by DyeVC in supporting developers and repository administrators. The study was conducted with four volunteers, which had previous experience with DVCS. They were graduate students from the Software Engineering research area at Universidade Federal Fluminense (UFF). Four sessions were conducted, each of them with one subject. The study was divided in two phases (without and with DyeVC), each one with two scenarios, where the subject had to answer questions related to usual work with DVCS. In Scenario 1, the subject played the developer role, working in a clone named *aakoch*. In Scenario 2, the subject played the repository administrator role. The following questions were posed: Q1.1 What is the status of your clone, compared to the central repository? Q1.2 Who else is working in the JQuery project (other clones)? Q1.3 Which files were modified in commit *5d454*? Q2.1 What are the existing clones for JQuery project? Q2.2 Which clones are synchronized with the central repository? Q2.3 How many commits in tracked branches are pending to be sent to the central repository? Q2.4 Is there any commit in non-tracked branches? Where?

In Phase 1 (without DyeVC), DyeVC was not in place and the subject answered the questions using any desired DVCS client among the ones available in the computer used in the experiment: *gitk, Tortoise Git, Git Bash*, and *SourceTree*. Participants were allowed to access the Internet and search any other procedure or

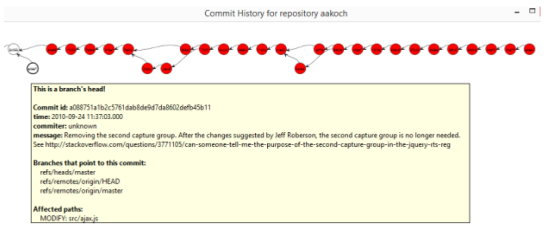


| Status | System Name | Clone Name | ID | Clone Path |
|---|---|---|---|---|
| ✓ | jquery | aakoch | rep1403040908134 | F:\evaluation\repos\aakoch |
| ⇅ | jquery | jeresig | rep1403042531580 | F:\evaluation\repos\jeresig |
| ↓ | jquery | adam | rep1403042549624 | F:\evaluation\repos\adam |

**ID:** rep1403042549624
**Address:** F:\evaluation\repos\adam
**Last checked on:** 2014-06-17 19:21:29

**Behind:** 121
**Local branch:** master **Remote branch:** refs/remotes/origin/master
**Remote url:** F:/evaluation/repos/central-repo

**Figure 10. Adam's tracked branches**



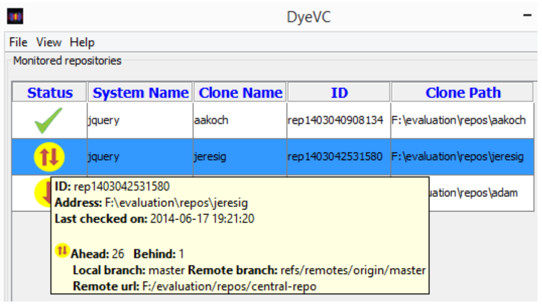
**This is a branch's head!**

**Commit id:** a088751a1b2c5761dab8de9d7da8602defb45b11
**time:** 2010-09-24 11:37:03.000
**commiter:** unknown
**message:** Removing the second capture group. After the changes suggested by Jeff Roberson, the second capture group is no longer needed. See http://stackoverflow.com/questions/3771105/can-someone-tell-me-the-purpose-of-the-second-capture-group-in-the-jquery-rts-reg

**Branches that point to this commit:**
refs/heads/master
refs/remotes/origin/HEAD
refs/remotes/origin/master

**Affected paths:**
MODIFY: src/ajax.js

**Figure 11. Aakoch's commit history**



| Status | System Name | Clone Name | ID | Clone Path |
|---|---|---|---|---|
| ✓ | jquery | aakoch | rep1403040908134 | F:\evaluation\repos\aakoch |
| ⇅ | jquery | jeresig | rep1403042531580 | F:\evaluation\repos\jeresig |
| ↓ |  |  |  | ation\repos\adam |

**ID:** rep1403042531580
**Address:** F:\evaluation\repos\jeresig
**Last checked on:** 2014-06-17 19:21:20

**Ahead:** 26 **Behind:** 1
**Local branch:** master **Remote branch:** refs/remotes/origin/master
**Remote url:** F:/evaluation/repos/central-repo

**Figure 12. Jeresig's tracked branches**

**Table 3. Time spent (in minutes) to answer each question**

| *Subject* | *Scenario 1* | | *Scenario 2* | |
|---|---|---|---|---|
| | *Phase 1* | *Phase 2* | *Phase 1* | *Phase 2* |
| P1 | 14 | 5 | - | 6 |
| P2 | 13 | 6 | - | 5 |
| P3 | 3 | 2 | - | 4 |
| P4 | 10 | 2 | - | 10 |

tool that could help in answering the questions. After that, the subject watched a 10-minute video presenting DyeVC and started Phase 2 (with DyeVC), which consisted in answering the same questions with the help of DyeVC. The possible answers in Phase 2 were either "keep the answer of Phase 1", meaning that using DyeVC did not change the subject perception, or a different answer, otherwise.

Table 3 presents the time spent by each subject to answer each question in both scenarios and both phases. It is possible to notice, by looking at Table 3, that all subjects took less time to complete Scenario 1 (developer role) in Phase 2 (with DyeVC). For Scenario 2 (admin role), times for Phase 1 (without DyeVC) are not shown because none of the subjects managed to answer the questions without using DyeVC.

**Table 4. Scalability results of DyeVC for repositories with different sizes**

| *Repository* | *Hosting* | *Repository metrics* | | | *Foreground operations* | | | *Background operations times (s)* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | *# commits* | *Size (MB)* | *# files* | *Commit History* | | *Topology* | *Insert 1st* | *Insert 2nd* | *Check Branches* | *Update Topology* |
| | | | | | *Time (s)* | *Memory Usage** | *Time (s)* | | | | |
| DyeVC | github.com | 187 | 1.0 | 539 | 3.5 | 15 | 2.7 | 12.4 | 16.1 | 1.7 | 4.4 |
| SAPOS | github.com | 702 | 7.0 | 685 | 5.6 | 19 | 3.2 | 20.8 | 22.6 | 1.8 | 5.2 |
| jgit | eclipse.org | 2,979 | 10.0 | 1,595 | 18.4 | 512 | 3.4 | 42.4 | 46.0 | 5.9 | 6.8 |
| egit | eclipse.org | 3,775 | 27.0 | 1,478 | 21.3 | 559 | 3.7 | 49.6 | 46.6 | 4.2 | 7.3 |
| jquery | github.com | 5,518 | 20.0 | 253 | 65.0 | 1,121 | 4.1 | 40.0 | 37.4 | 1.4 | 9.4 |
| Tortoise Git | code.google.com | 6,166 | 85.0 | 3,220 | 68.0 | 492 | 4.2 | 39.0 | 36.0 | 1.6 | 9.6 |
| Gitextensions | github.com | 6,417 | 448.0 | 1,549 | 73.0 | 1,529 | 17.0 | 155.8 | 129.0 | 1.6 | 10.6 |
| Drupal | drupal.org | 23,922 | 84.4 | 9,290 | - | - | 18.0 | 102.0 | 95.0 | 2.0 | 18.0 |
| Expresso Livre | gitorious.org | 25,822 | 141.0 | 20,729 | - | - | 18.2 | 110.0 | 102.0 | 2.1 | 19.3 |
| Git | github.com | 35,260 | 98.0 | 2,656 | - | - | 19.4 | 196.0 | 158.6 | 3.4 | 40.0 |

* Memory usage was measured in MB during the execution of "Commit History" operation.

**Table 5. Spearman's rank correlation coefficient between repository size metrics and DyeVC operations time**

| *Operation* | *# commits* | *Size* | *# files* |
|---|---|---|---|
| Insert 1st | 0.85 | 0.83 | 0.76 |
| Insert 2nd | 0.85 | 0.83 | 0.76 |
| Check Branches | 0.07 | -0.05 | 0.72 |
| Update Topology | 1.00 | 0.88 | 0.52 |
| Commit History | 1.00 | 0.96 | -0.04 |
| Topology | 1.00 | 0.88 | 0.52 |

The overall results of this study were positive. In Phase 1 (without DyeVC), each subject used a different tool and followed a different procedure to find answers regarding DVCS usage. Subjects were able to answer correctly questions Q1.1 and Q1.3 whether using DyeVC or not. In addition, further questions were answered correctly only by using DyeVC.

The subjects also answered an exit questionnaire. All subjects found easy to interact with DyeVC, to identify related repositories, and to use the operations available. They consensually elected the topology visualization as the most helpful visualization in DyeVC. In addition, by using the Product Reaction Cards[5], 3 out of the 4 subjects stated that DyeVC is helpful and easy to use.

## 5.3 Performance Study

In order to evaluate the scalability of our approach, we measured the time spent to perform the most common DyeVC operations. We used projects of different sizes and hosted in different Git servers. Table 4 shows the monitored projects (name and hosting service), the repository metrics – number of commits, disk usage, and number of files – and the time spent to run some background and foreground operations in DyeVC. All measurements were taken in the same period of the day and from the same machine, a Core Duo CPU at 2.53 GHz, with 4GB RAM running Windows 8.1 Professional 64 bits, connected to the internet at 35 Mbit/s.

We measured the main operations of our approach: "Insert 1st", invoked when the user includes the first repository of a given system to be monitored; "Insert 2nd", invoked when the user includes a repository to be monitored in a system that already have registered repositories; "Commit History", invoked when the user requests to see the commit history of a given repository; "Topology", invoked when the user wants to see the topology of repositories of a given system; "Check Branches", invoked periodically to check all the monitored repositories, searching for ahead or behind commits; and "Update Topology", invoked periodically to update the topology information in the central database. This last operation updates the existing repositories, their peers, and the existing commits, marking in which repositories each commit is found, as detailed by Algorithm 1.

[5] Developed by and © 2002 Microsoft Corporation. All rights reserved.

It is possible to notice that the "Commit History" operation has no values for the last three repositories. This occurs because, as the number of commits increases, more memory is used to calculate the commit history graph. The current algorithm has an $O(x^2)$ space complexity (being $x$ the number of commits). Our experiment computer was configured with a 2 GB maximum Java Heap Size, which let us analyze repositories with up to 6K commits. This limitation occurs mainly because of JUNG.

Table 5 shows the correlation between each repository size metric and the DyeVC operations execution time, according to the Spearman's rank correlation coefficient [26]. This correlation coefficient measures the monotonic relation between two variables and ranges from *-1* to *1*. Values of *1* or *-1* mean that each variable is a perfect (increasing or decreasing) monotone function of the other. A value of *0* means that there is no correlation between the variables.

Looking at Table 5, it is possible to notice that, except for the "Check Branches" operation, all other operation times are strongly correlated to the number of commits and repository size. This is due to the nature of these operations, which update or show information about all commits in the repository. On the other hand, except for the "Commit History" operation, all other operation times correlate with the number of files. This is also expected due to the nature of "Commit History" operation, which does not dig into the changed files.

## 5.4 Threats to Validity

While we have taken care to minimize threats to the validity of the experiment, some factors can influence the results. The usage of a *post-hoc* analysis to evaluate a real project may not reflect the exact sequence of events that occurred, although the outcome did not change. For example, when we say that *aakosh*, at some moment, had 121 commits pending to be pushed to the central repository, these commits could have been pushed at once or by a series of smaller pushes. Moreover, only one project was selected to perform the analysis, what imposes limitations from a generalization standpoint. Furthermore, we used an open source project to perform the *post-hoc* analysis, but the *modus operandi* of peers may be different in academic or industrial contexts.

In the observation study, the selection of subjects was done by asking for volunteers from students in the same research group of the author. This was necessary due to time and people restrictions. Therefore, this group might not be representative and can be biased. Moreover, there were few subjects in this study. Thus, the results may have been influenced by the size and by specific characteristics of the group. Furthermore, subjects performed tasks involving DyeVC right after knowing the approach, giving no time to subjects to assimilate the tool. Results may have been influenced by this lack of time to mature the necessary knowledge to use the approach efficiently. In addition, subjects could have answered questions in Phase 2 faster than in Phase 1 due to their learning regarding the scenario.

Finally, there is a risk regarding the instrumentation used to measure the response times during the performance evaluation. As we used a database stored over the Internet, connectivity issues and network instability may have affected the response times.

## 6. RELATED WORK

According to Diehl [10], software visualization can be separated into three aspects: structure, behavior, and evolution. DyeVC relates primarily with the evolution aspect, more specifically with studies that aim at improving the awareness of developers that work with distributed software development. A recent work by Steinmacher et al. [27] presents a systematic review of awareness studies, which we used to perform a forward and backward snowballing. The approaches obtained after the snowballing were divided into four groups. The first group includes approaches that notify commit activities. The second group comprises approaches that not only give the developer awareness of concurrent changes, but also inform them about conflicts. The third group includes approaches that visualize repository information. Finally, the fourth group contains commercial and open source DVCS clients.

The first group contains tools such as SVN Notifier[6], SCM Notifier[7], Commit Monitor[8], SVN Radar[9], Hg Commit Monitor[10] and Elvin [15]. The primary focus of these approaches is on increasing the developer's perception of concurrent work by showing notifications whenever other developers perform actions. The approaches in this group do not identify related repositories and do not provide information in different levels of details, such as status, branches, and commits. DyeVC provides these different levels of details, as shown in Section 3.2.

The second group comprises approaches that give the developer awareness of concurrent changes, sometimes informing them if conflicts are likely to occur. This group includes tools such as Palantir, [24], CollabVS [9], Crystal [3], Lighthouse [25], FASTDash [2], and WeCode [17]. Among these, only Crystal and FASTDash work with DVCSs. Crystal detects physical, syntactic, and semantic conflicts in Mercurial and Git repositories (provided that the user informs the compiling and testing commands), but does not precisely deal with repositories that pull updates from more than one peer. FASTDash does not detect conflicts directly, as the previous cited studies, but provides awareness of potential conflicts, such as two programmers editing the same region of the same source file in repositories stored in Microsoft Team Foundation Server. Although DyeVC primary focus is not to detect conflicts, it can be combined with such approaches to allow conflicts and metrics analysis over DVCS.

The third group includes approaches that visualize repository information. Each approach has a different visualization focus, such as program structures [7], classes [19], lines [29], authors [16], and branch history [12][11, 12]. The latter have the same focus of DyeVC's Commit History visualization, but dealing only with the local repository, not showing, for example, where a given commit can be found in related repositories.

Finally, the fourth group includes commercial / open source DVCS clients, which allows one to execute operations on repositories / clones (push, pull, checkout, commit, etc.) and also visualizing the repository history, i.e., the commits, along with their attributes (comment, date, affected files, committer, etc.). For example, some Git clients include *gitk*[13], *TortoiseGit*[14], *EGit for Eclipse*[15], and *SourceTree*[16]. The data about commits shown by these tools varies, but generally involves the committer name, message, date, affected files, and a visual representation of the history. These tools, though, have no knowledge regarding peers. For this reason, they do not present commits from other clones and do not include information about where each commit can be found. It is worth noticing that we could not find any similar work showing the dependencies among several clones of a DVCS.

All in all, among related work, *Crystal* is the most similar to DyeVC, and deserves a deeper comparison. Both approaches work with DVCSs (besides Git, *Crystal* also supports Mercurial) and use working copies to perform analyses, but there are major differences between them. *Crystal's* **goal** is to identify conflicts among pairs of repositories, whereas *DyeVC's* goal is to provide awareness regarding the existing peers and their synchronization, in different levels. To **identify repositories**, *Crystal* demands the user to point out all repositories they want to compare, whereas *DyeVC* requires that some of the repositories be registered and it automatically looks at configuration files to discover all the repositories that one pushes to or pulls from. The **repository comparison** in *Crystal* is from one repository against all the other together, whereas *DyeVC* analyzes each repository against each other, providing a pairwise view and a combined view of the history. Finally, the **allowed actions** in *Crystal* include the ability to *push*, *pull*, *compile*, and *test* a repository, whereas *DyeVC* allows one to visualize branches status, topology, and history. In this way, we see potential to have both tools working together to better provide awareness and safety when working with DVCS.

## 7. CONCLUSION

In this paper we presented DyeVC, an approach that identifies the status of a repository in contrast with its peers, which are dynamically found in an unobtrusive way. We have evaluated DyeVC on a real project, showing that it can be used to answer questions that arise when working with DVCSs. The observational study results were promising: DyeVC was considered easy to use and fast for most repository history exploration operations, while providing the expected answers. This provides initial evidence that DyeVC could effectively help developers and repository administrators by saving time and by supporting answering questions regarding DVCS usage that could not be answered before. We have also evaluated DyeVC's performance over repositories of different sizes, and we found out that the time and space complexity of the approach are directly related to the number of commits in the repository, especially in the view levels with finer granularity.

[6] http://svnnotifier.tigris.org/ (2012)
[7] https://github.com/pocorall/scm-notifier (2012)
[8] http://tools.tortoisesvn.net/CommitMonitor.html (2013)
[9] http://code.google.com/p/svnradar/ (2011)
[10] http://www.fsmpi.uni-bayreuth.de/~dun3/hg-commit-monitor (2009)

[11] Visugit: https://github.com/hozumi/visugit
[12] GitHub's Network Graph: https://github.com/blog/39-say-hello-to-the-network-graph-visualizer
[13] http://git-scm.com/docs/gitk
[14] https://code.google.com/p/tortoisegit/
[15] http://eclipse.org/egit/
[16] http://www.sourcetreeapp.com/

A number of future researches arise from this work. Different visualizations can be developed to show the commit history, compacting it, for example, by automatically collapsing contiguous nodes representing commits with the same level of accessibility. DyeVC could gather additional metadata, for example, to create a visualization showing conflicts that would happen when merging two or more branches. This data could also be used to mine information in the repositories, revealing usage patterns or presenting metrics. Finally, some optimization should be done to allow DyeVC work with larger repositories.

## 8. ACKNOWLEDGMENTS

We thank CNPq and FAPERJ for the financial support.

## 9. REFERENCES

[1] Appleton, B., Berczuk, S., Cabrera, R. and Orenstein, R. 1998. Streamed lines: Branching patterns for parallel software development. *Pattern Languages of Programs Conference (PLoP)* (Monticello, Illinois, USA, Aug. 1998).

[2] Biehl, J.T., Czerwinski, M., Smith, G. and Robertson, G.G. 2007. FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams. *ACM Conference on Human Factors in Computing Systems (CHI)* (San Jose, California, USA, May 2007), 1313–1322.

[3] Brun, Y., Holmes, R., Ernst, M.D. and Notkin, D. 2011. Proactive detection of collaboration conflicts. *ACM SIGSOFT Symposium and European Conference on Foundations of Software Engineering (ESEC/FSE)* (Szeged, Hungary, Sep. 2011), 168–178.

[4] Cederqvist, P. 2005. *Version Management with CVS*. Free Software Foundation.

[5] Cesario, C.M. and Murta, L.G.P. 2013. What is going on around my repository? *Brazilian Workshop on Software Visualization, Evolution and Maintenance (VEM)* (Brasilia, Brazil, Sep. 2013), 14–21.

[6] Chacon, S. 2009. *Pro Git*. Apress.

[7] Collberg, C., Kobourov, S., Nagra, J., Pitts, J. and Wampler, K. 2003. A System for Graph-based Visualization of the Evolution of Software. *ACM Symposium on Software Visualization (SOFTVIS)* (San Diego, CA, USA, Jun. 2003), 77–ff.

[8] Collins-Sussman, B., Fitzpatrick, B.W. and Pilato, C.M. 2011. *Version Control with Subversion*. Compiled from r4849.

[9] Dewan, P. and Hegde, R. 2007. Semi-synchronous conflict detection and resolution in asynchronous software development. *European Conference on Computer-Supported Cooperative Work (ECSCW)* (Limerick, Ireland, Sep. 2007), 159–178.

[10] Diehl, S. 2007. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer.

[11] Eclipse Foundation 2014. *2014 Annual Eclipse Community Report*. Eclipse Foundation.

[12] Elsen, S. 2013. VisGi: Visualizing Git branches. *IEEE Working Conference on Software Visualization (VISSOFT)* (Eindhoven, Netherlands, Sep. 2013), 1–4.

[13] Estublier, J. 2000. Software configuration management: a roadmap. *Internation Conference on Software Engineering (ICSE)* (Limerick, Ireland, May 2000), 279–289.

[14] Fielding, R.T. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. University of California.

[15] Fitzpatrick, G., Marshall, P. and Phillips, A. 2006. CVS Integration with Notification and Chat: Lightweight Software Team Collaboration. *ACM Conference on Computer-supported Cooperative Work (CSCW)* (Banff, Alberta, Canada, Nov. 2006), 49–58.

[16] Gilbert, E. and Karahalios, K. 2006. LifeSource: Two CVS Visualizations. *ACM Conference on Human Factors in Computing Systems (CHI)* (Montreal, Canada, Apr. 2006), 791–796.

[17] Guimarães, M.L. and Silva, A.R. 2012. Improving early detection of software merge conflicts. *Internation Conference on Software Engineering (ICSE)* (Zürich, Switzerland, Jun. 2012), 342–352.

[18] Gumm, D.-C. 2006. Distribution Dimensions in Software Development Projects: A Taxonomy. *IEEE Software*. 23, 5 (Sep. 2006), 45–51.

[19] Lanza, M. 2001. The Evolution Matrix: Recovering Software Evolution Using Software Visualization Techniques. *International Workshop on Principles of Software Evolution (IWPSE)* (Tokyo, Japan, Sep. 2001), 37–42.

[20] O'Sullivan, B. 2009. Making sense of revision-control systems. *Communications of the ACM*. 52, 9 (Sep. 2009), 56–62.

[21] O'Sullivan, B. 2009. *Mercurial: The Definitive Guide*. O'Reilly Media.

[22] Perry, D.E., Siy, H.P. and Votta, L.G. 1998. Parallel changes in large scale software development: an observational case study. *International Conference on Software engineering (ICSE)* (Kyoto, Japan, Apr. 1998), 251–260.

[23] Rochkind, M.J. 1975. The source code control system. *IEEE Transactions on Software Engineering (TSE)*. 1, 4 (Dec. 1975), 364–470.

[24] Sarma, A. and van der Hoek, A. 2002. Palantir: coordinating distributed workspaces. *26th Computer Software and Applications Conference (COMPSAC)* (Oxford, United Kingdom, Aug. 2002), 1093 – 1097.

[25] da Silva, I.A., Chen, P.H., Van der Westhuizen, C., Ripley, R.M. and van der Hoek, A. 2006. Lighthouse: coordination through emerging design. *Workshop on Eclipse Technology eXchange (ETX)* (Portland, Oregon, USA, Oct. 2006), 11–15.

[26] Spearman, C. 1904. The Proof and Measurement of Association between Two Things. *The American Journal of Psychology*. 15, 1 (1904), 72–101.

[27] Steinmacher, I., Chaves, A. and Gerosa, M. 2012. Awareness Support in Distributed Software Development: A Systematic Review and Mapping of the Literature. *ACM Conference on Computer-supported Cooperative Work (CSCW)* (Seattle, WA, USA, May 2012), 1–46.

[28] Tichy, W. 1985. RCS: A system for version control. *Software - Practice and Experience*. 15, 7 (1985), 637–654.

[29] Voinea, L., Telea, A. and van Wijk, J.J. 2005. CVSscan: Visualization of Code Evolution. *ACM Symposium on Software Visualization (SOFTVIS)* (Saint Louis, MO, USA, May 2005), 47–56.

[30] Walrad, C. and Strom, D. 2002. The importance of branching models in SCM. *IEEE Computer*. 35, 9 (Sep. 2002), 31 – 38.

[31] Wloka, J., Ryder, B., Tip, F. and Ren, X. 2009. Safe-commit analysis to facilitate team software development. *International Conference on Software Engineering (ICSE)* (Vancouver, British Columbia, Canada, May 2009), 507–517.