Journal of Software Engineering Research and Development

RESEARCH Open Access

DyeVC: an approach for monitoring and visualizing distributed repositories

Cristiano Cesario, Ruben Interian and Leonardo Murta*

* Correspondence:
leomurta@ic.uff.br
Instituto de Computação, Universidade Federal Fluminense (UFF), Niteroi, RJ, Brazil

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 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 and a performance evaluation 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.

Keywords: Distributed version control, Monitoring, Visualization, Awareness

1 Background

Version Control Systems (VCS) date back to the 70s when SCCS emerged (Rochkind 1975). Their primary purpose is to keep software development under control (Estublier 2000). Along these almost 40 years, VCSs have evolved from a centralized repository with local access (e.g., SCCS and RCS (Tichy 1985)) to a client-server architecture (e.g., CVS (Cederqvist 2005) and Subversion (Collins-Sussman et al. 2011)). More recently, distributed VCSs (DVCS) arose (e.g., Git (Chacon 2009) and Mercurial (O'Sullivan 2009a)) allowing clones of the entire repository in different locations. According to a survey conducted by the Eclipse community (2014), 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 (Walrad and Strom 2002), creating branches in VCSs is essential to software development because it enables parallel 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 (O'Sullivan 2009b), 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 (Gumm 2006), 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. (1998), concurrent development increases the number of defects in software. Besides, da Silva et al. (2006) 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. (2011) 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. (1998), 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 on 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 (Sarma and van der Hoek 2002), FASTDash (Biehl et al. 2007), Lighthouse (da Silva et al. 2006), CollabVS (Dewan and Hegde 2007), Safe-Commit (Wloka et al. 2009), Crystal (Brun et al. 2011), and WeCode (Guimarães and Silva 2012), 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.

To answer the questions above, we propose 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. 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 and a performance evaluation 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.

This paper extends a previous conference paper (Cesario and Murta 2016) by including a more thorough discussion of our approach, including how DyeVC discovers the topology and a formal definition of the process underneath DyeVC. Moreover, as the previous version of DyeVC struggled when dealing with large repositories (over 6500 commits), we also added an automatic collapsing feature. This new feature provides a dual contribution: it allowed DyeVC to deal with larger repositories and reduced cluttering when presenting information to users. The performance evaluation was expanded to present an assessment of the automatic collapsing feature. Finally, we included a deeper comparison of DyeVC with its related work. This paper is organized as follows: Section 2 shows a motivational example. 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 suggestions for future work.

2 Motivational example

Figure 1 shows a scenario with some developers, each one owning a clone of the repository 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 Fig. 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.

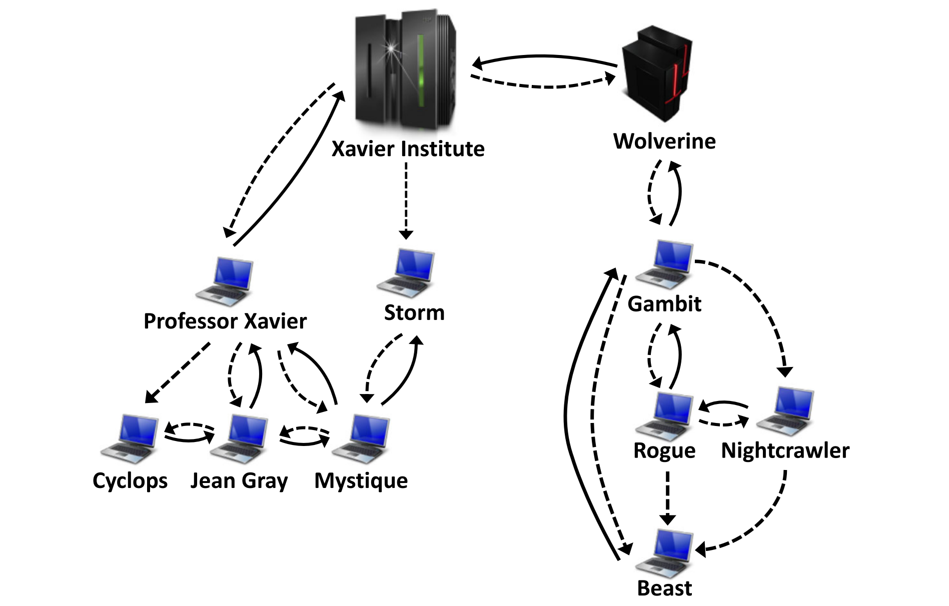


Fig. 1 Development scenario involving some developers

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, questions such as the following can arise:

- 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? Suppose that Beast is working on a feature that depends on a utility class developed by Gambit. If such class has a bug, and Gambit is working to solve it, Beast would want to know when Gambit's commit is ready to be pulled. Moreover, if Gambit is evolving such class, and Beast has this information, he could decide to anticipate a pull to incorporate Gambit's changes in his workspace.
- Would it be the case that local commits are 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, questions such as the following are pertinent:

- How can she knows which are the existing clones of a project and how they relate to each other? This is a common need to repository administrators. It helps not only in identifying who must be notified regarding any news related to the repository but also helps in visually verifying if pull/push policies are being followed by the team. Having a map of all existing clones can help repository administrators in identifying who is pushing to / pulling from each other. For instance, unauthorized access to push to a production repository can be visualized, and the administrator can take actions to revoke such access.
- How can she know if there are pending commits to be sent from a staging repository to a production one? Having the ability to know how many commits are pending and which commits are these can help administrators decide if this is the right time to release a new version of the system to production.

3 DyeVC approach

Aiming at supporting both developers and repository administrators in understanding the interaction among repository clones, the main features of DyeVC include: (1) a mechanism to gather information from a set of clones (such as their relationships and known commits) 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. Figure 2 shows a deployment view of DyeVC's architecture. For each clone *rep* that the user registers to monitor, DyeVC transparently creates a local clone *rep'* in the user's home folder to fetch data from all of the peers with which *rep* communicates. 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 data or to pull data), we can 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.

Figure 3 shows how DyeVC discovers the topology from the nodes where it is running and the registered clones. Blue nodes represent registered clones where DyeVC is running, yellow nodes represent known clones located at nodes where DyeVC is not running, dashed nodes and dashed lines represent clones and communication paths, respectively, that are not known yet. Suppose a scenario where the existing clones and interdependencies are shown in Fig. 3a, which depicts the same scenario shown in Section 2 but here represented by the first letter of each clone. After installing DyeVC and

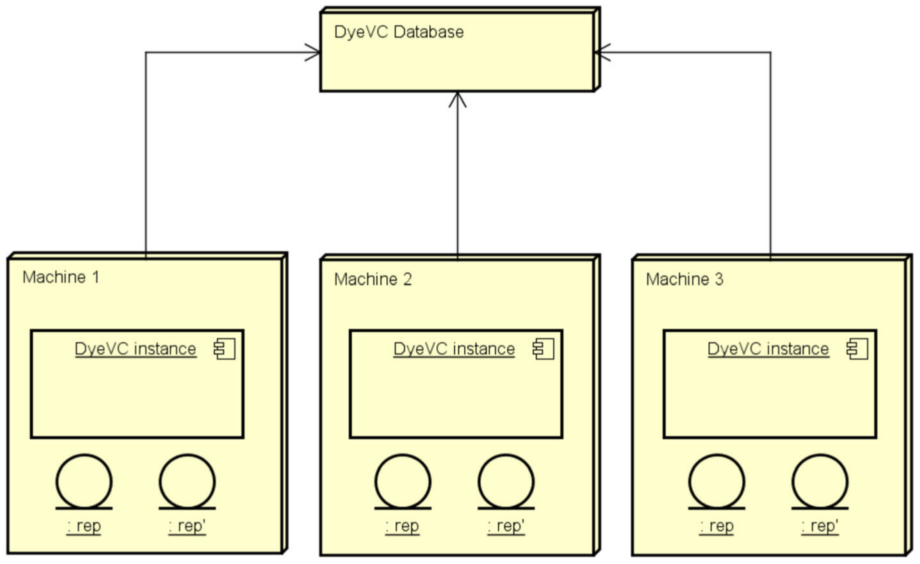


Fig. 2 How DyeVC gathers information

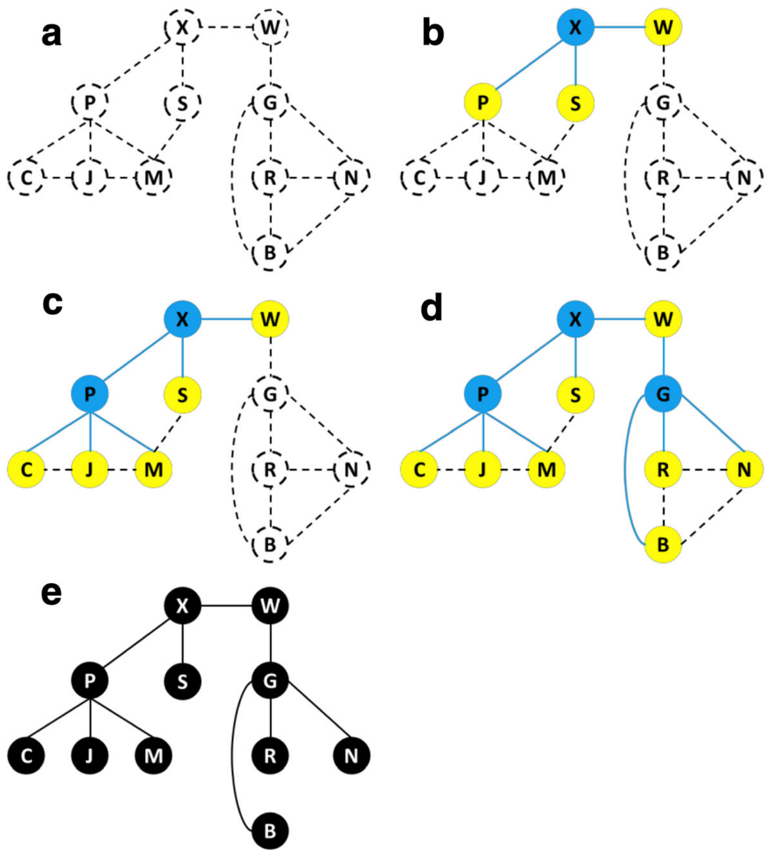
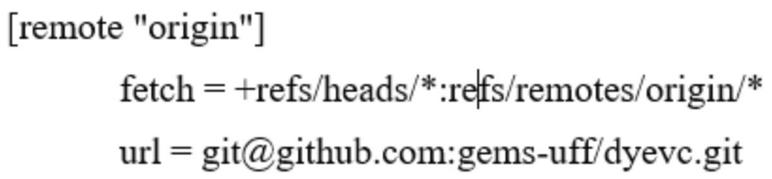


Fig. 3 DyeVC discovering the topology: actual topology (**a**), discovered topology with X (**b**), discovered topology with X and P (**c**), discovered topology with X P and G (**d**), final discovered topology (**e**)

registering clone *X*, DyeVC finds out that this clone communicates with clones *W*, *P*, and *S* (either by pushing to or pulling from them), as shown in Fig. 3b. Later on, clone *P* is registered and clones *C*, *J* and *M* are included as known clones in the topology (Figure 3c). Clone *G* is the next to be registered, allowing DyeVC to discover that clones *R*, *N*, and *B* also exist, as well as the communication between clone *G* and clone *W*, which was already a known clone (Fig. 3d). Assuming that no other clones are registered, the known topology is shown in Fig. 3e. Notice that, although only clones *G*, *P*, and *X* were registered, DyeVC is also aware of the existence of clones *B*, *C*, *J*, *M*, *N*, *S* and *W*. Only some communication paths between clones will not be known (*C-J*, *J-M*, *S-M*, *R-B*, *R-N* and *N-B*).

DyeVC finds out related clones by looking at the remote repositories, which are registered in Git's *config* file of each clone. Figure 4 shows an example of this configuration, taken from a local clone of the *DyeVC* project, where there is a remote named *origin*, which is located at *github.com/gems-uff/dyevc*. This information is in the *url* parameter, which indicates to Git that pushes and pulls use the same location. If there were a *pushurl* parameter in the configuration, besides the *url* parameter, pulls would use the location in the *url* parameter and pushes would use the location in the *pushurl* parameter.

Data stored in the central database follows the metamodel presented in Fig. 5. A *Project* groups repository clones of the same system. Clones are stored as *RepositoryInfo*



```
[remote "origin"]
        fetch = +refs/heads/*:refs/remotes/origin/*
        url = git@github.com:gems-uff/dyevc.git
```

Fig. 4 Remote repository configuration in Git's *config* file

and are identified by an *id* and a meaningful clone name provided by the user (*cloneName* attribute). 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), its *clonePath* (be it an operating system path or an URL) and the set of DyeVC instances that have registered it to be monitored (*monitoredBy* attribute).

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 (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 (*headOf* association end). The second association represents which commits are reachable from a given branch (*reachableCommits* association end) and, conversely, the branches from which the commit is reachable (*reachableFrom* association end).

The finer grain of information is the *CommitInfo*, which represents each commit in the topology. A commit is identified by a hash code (*hash* attribute) and refers to its

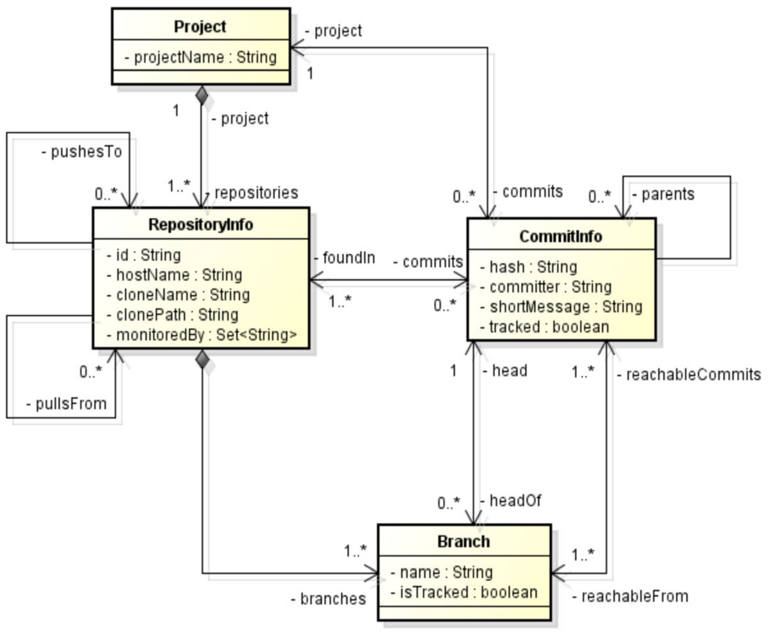


Fig. 5 Metamodel used to store DyeVC data

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 end). We also store the *committer*, the commit message (*shortMessage* attribute), and whether the commits belong to tracked or non-tracked branches (*tracked* attribute).

3.2 Information visualization

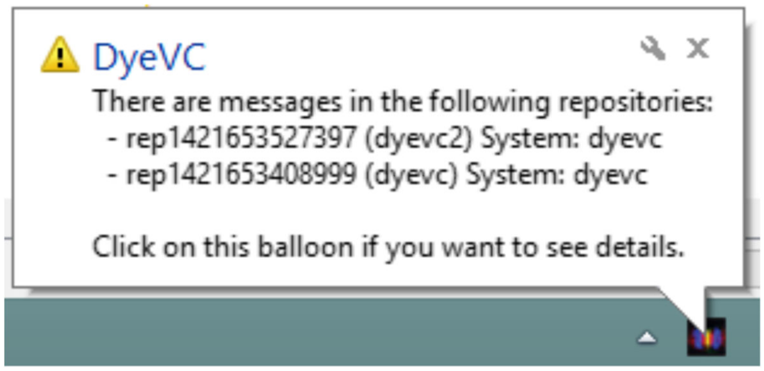
DyeVC presents information at four different levels of detail: Level 1 shows high-level notifications about registered repositories; Level 2 shows the whole topology of a given project. Level 3 zooms into the branches of the repository, showing the status of each tracked branch. Lastly, Level 4 zooms 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 6 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 to answer questions Q1 and Q2, we present a topology view showing all repositories for a given project (Fig. 7), 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. At first sight, this could be understood as a risk within topology view. However, DyeVC considers servers as clones. The denomination "server" is just to visually differentiate it from other clones. We believe that plotting servers and clones where


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.

Fig. 6 DyeVC showing notifications in notification area

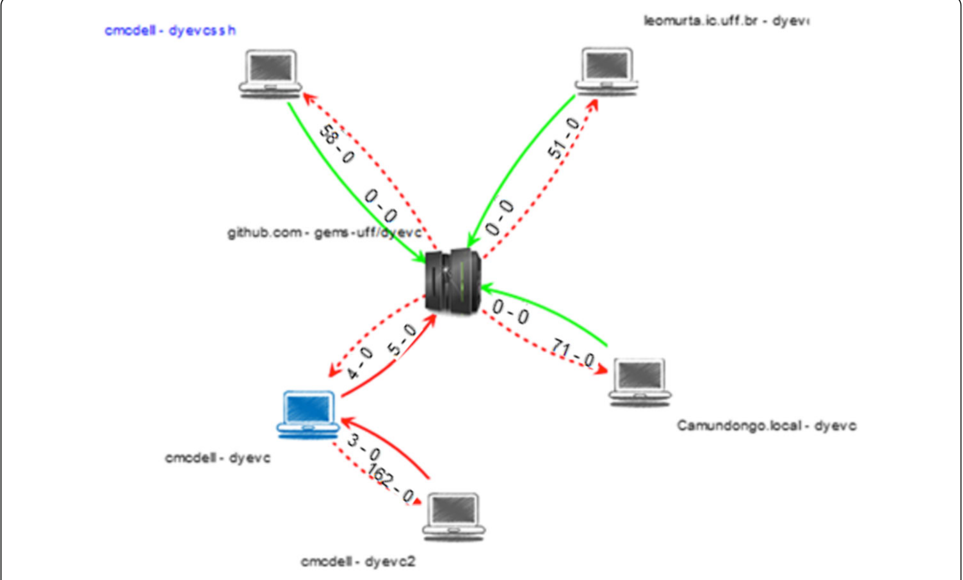


Fig. 7 Topology view for a given project

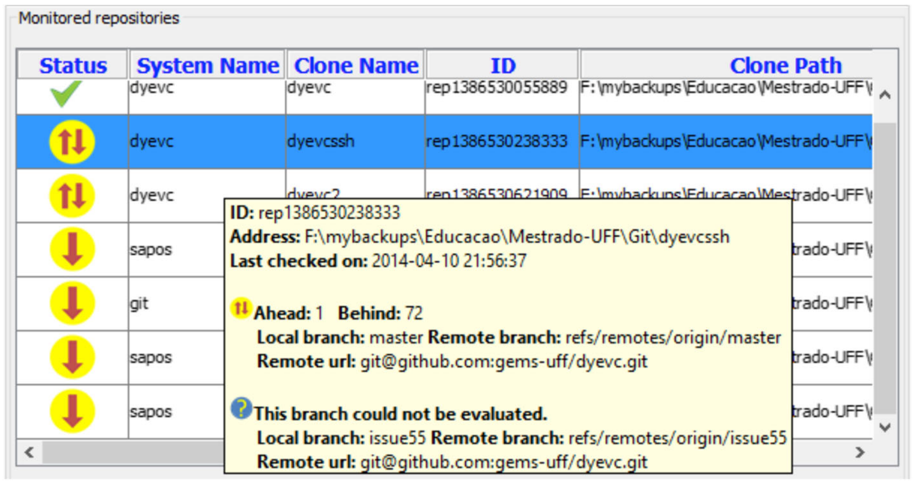
DyeVC is not running with the same icons is not a risk because the topology view brings more information about the clones (e.g., clone address and name). Thus, a repository administrator can distinguish the servers among the plotted clones.

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, respectively. 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 the commits present 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 Fig. 7 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. In this case, both edges are red, which raises attention to investigate further what is happening, because such situation may lead to integration conflicts.

3.2.3 Level 3: Tracked branches

For helping answering question Q3, DyeVC's main screen (see Fig. 8) shows Level 3 information, allowing one to view the status of each tracked branch in 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



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

Fig. 8 DyeVC main screen

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. In this case, local repository will be *behind* its peers (arrow down in Fig. 8). Conversely, subtracting the sets in the inverse order will result in the set of commits not pushed yet, meaning that local repository is ahead of its peers (arrow up). When both sets are empty, local repository is synchronized (green checkmark in Fig. 8) and when both sets have elements, it is both ahead and behind its peer (arrow up and down in Fig. 8).

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. Consider the synchronization paths presented in the right-hand side of Fig. 1, where the perception of each developer regarding 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). What happens is that a developer pulls from another (for example, between Gambit and Nightcrawler), avoiding 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 to provide an answer to Question Q3. Differently from the usual repository version graph, it presents a combined version graph of the entire topology (Fig. 9). Each vertex in the graph represents either a

Table 1 Existing commits in each repository

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

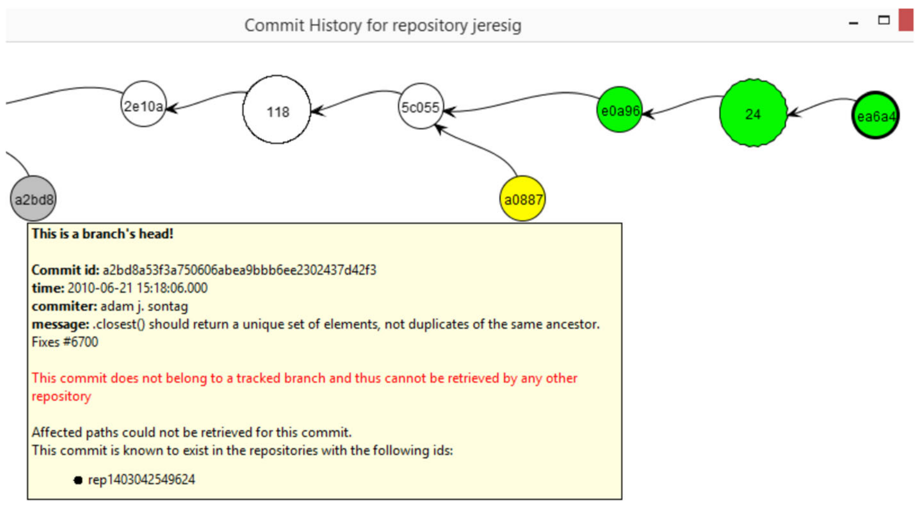
Table 2 Status of each repository based on known remote repositories

Repository	Wolverine	Gambit	Rogue	Nightcrawler	Beast
Wolverine	-	-	-	-	-
Gambit	✓	-	-	-	-
Rogue	-	⇅	-	-	-
Nightcrawler	-	✓	⇅	-	-
Beast	-	⬇	⬇	⬇	-

known commit in the topology, which is named after its hash's five first characters (e.g., the node labeled *2e10a* in Fig. 9), or a collapsed node, representing several commits blended. We implement two ways of collapsing nodes to provide a better understanding over huge amounts of data: manual and automatic. Manually collapsed nodes are named after the number of contained nodes, such as the white node containing 118 commits and the green node containing 24 commits in Fig. 9). Automatically collapsed nodes have ellipses before and after the number of contained nodes in their names (if the first collapse of Fig. 9 were automatic, its name would be "…118…"). Automatic collapsing is detailed in Section 3.2.5.

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 Fig. 5 (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 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



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

Fig. 9 Collapsed commit history

on 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 of 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 number of nodes, users are usually 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 Automatic collapsing

As previously discussed, the first version of DyeVC struggled when dealing with larger repositories (over 6500 commits). This limitation was mainly due to the memory used to represent commit nodes in the commit history graph. However, we observed that many of the commit nodes are unnecessary for comprehending the evolution and, in fact, were cluttering the visualization. For instance, a sequence of 20 commit nodes that are ordinary revisions and that belong to all clones (i.e., all have the same white color) could be collapsed into just one commit node, avoiding visualization cluttering and boosting performance. This observation motivated us to design and implement an automatic collapsing feature for DyeVC.

We identified two common node structures that can be automatically collapsed: sequential and parallel. The former contains a sequence of commits of the same type, where each of them has degree two, i.e., nodes with just one ancestor and one successor. This kind of structure can be collapsed because it does not represent any additional information besides the fact that some sequential work was performed. Figure 10 shows examples of sequences of commits, highlighted in red, which could be collapsed, producing the graph shown in Fig. 11 (still in red). On the other hand, the later contains one fork node and one merge node, with at most one (regular or collapsed) 2-degree node in each branch, between the fork and the merge nodes. Figure 11 shows examples highlighted in yellow of this parallel structure. The result of the collapse is shown in Fig. 12. The numbers inside the red and yellow circles refer to the number of collapsed nodes.

We implemented an iterative algorithm that works in phases to benefit from both sequential and parallel collapse strategies together. The algorithm is shown in Fig. 13.

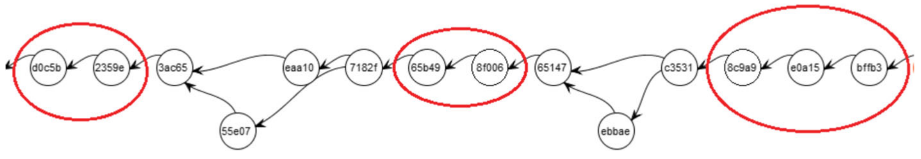


Fig. 10 Sequential structures before automatic collapsing

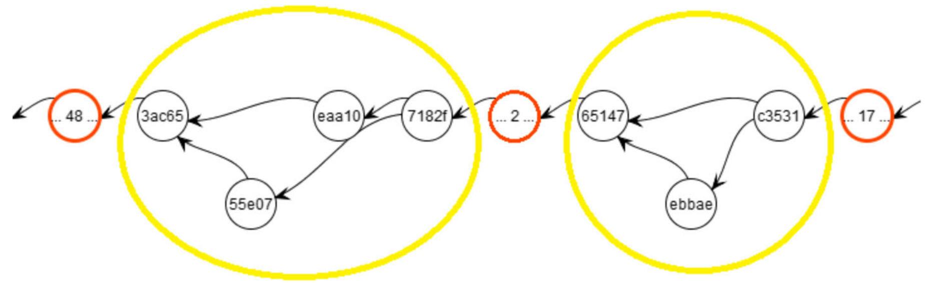


Fig. 11 Sequential structures after automatic collapsing and Parallel structures before automatic collapsing

The algorithm receives the commit graph and the number of iterations as parameters. Each iteration is executed in linear time complexity. The first phase collapses sequential structures (lines 2–11). The set of visited nodes is initialized as an empty set in line 2. Each node is inspected in a loop (lines 3–11). If the node is still not visited, the presence of a linear commit chain is tested in line 5 by examining predecessors and successors of the node. All found linear commit chain elements are marked as visited in line 6. If the linear commit chain has more than one node, it is collapsed (lines 7–9).

The second phase of the algorithm collapses parallel structures (lines 12–28). The *visited* set is reinitialized as an empty set in line 12. All nodes of the graph are examined again in a loop (lines 13–28). If the element is still not visited, and we are in the presence of a fork node (condition in line 14), the parallel structure generated by this fork is analyzed. Both nodes afterward the fork are saved in variables *a* and *b* (lines 15–16). Initially, a *group* set is initialized containing a single element, the fork node, in line 17. If the parallel structure resembles the 4-node group highlighted in yellow in Fig. 11, then the group to be collapsed is populated in lines 18–19. On the other hand, if the parallel structure resembles the 3-node group highlighted in yellow in Fig. 11, then the group to be collapsed is populated in lines 20–21. The visited set is updated in line 23. If the created node group has more than one node, it is collapsed in lines 24–26.

The phases of the algorithm can be repeated, as collapsing parallel structures may lead to new sequential structures. For instance, after applying parallel collapses over the graph shown in Fig. 11, a new sequential structure is formed, as illustrated in Fig. 12. The iteration would lead to a new collapse, and so on. As previously discussed, collapses are performed just for commits of the same type (same color, discussed in section 3.2.4), reducing the size of the graph without compromising the quality of the information shown in the graph.

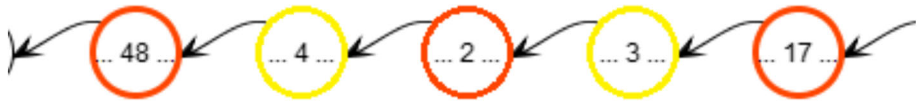


Fig. 12 Parallel structures after automatic collapsing

```
Algorithm 1 Automatic Collapse of Commit Graph.
Input:
    $graph$                                        ▷ The commit graph
    $iterations$                    ▷ The number of collapse iterations
while $iterations > 0$ do
    $visited \leftarrow \emptyset$
    for each $node \in graph.nodes$ do           ▷ Sequential Collapse
        if $node \notin visited$ then
            $group \leftarrow chain(node, OUT) \cup \{node\} \cup chain(node, IN)$
            $visited \leftarrow visited \cup group$
            if $|group| > 1$ then
                $collapse(group)$
            end if
        end if
    end for
    $visited \leftarrow \emptyset$
    for each $node \in graph.nodes$ do             ▷ Parallel Collapse
        if $node \notin visited \wedge |node.in| = 2$ then
            $a \leftarrow node.in[0]$
            $b \leftarrow node.in[1]$
            $group \leftarrow \{node\}$
            if $linear(a) \wedge linear(b) \wedge a.in[0] = b.in[0]$ then
                $group \leftarrow group \cup \{a, b, a.in[0]\}$
            else if $(linear(a) \wedge a.in[0] = b) \vee (linear(b) \wedge b.in[0] = a)$ then
                $group \leftarrow group \cup \{a, b\}$
            end if
            $visited \leftarrow visited \cup group$
            if $|group| > 1$ then
                $collapse(group)$
            end if
        end if
    end for
    $iterations \leftarrow iterations - 1$
end while
return $graph$.

function CHAIN(node,direction)
    $chain \leftarrow \emptyset$
    while $linear(node)$ do
        $chain \leftarrow chain \cup node$
        if direction=IN then
            $node \leftarrow node.in[0]$
        else
            $node \leftarrow node.out[0]$
        end if
    end while
    return $chain$
end function

function LINEAR(node)
    return $|node.in| = 1 \wedge |node.out| = 1$
end function
```

Fig. 13 Automatic collapsing algorithm

3.4 Behind the scenes

The process underneath DyeVC can be formally defined using Set Theory, in order to describe how the data is structured and how DyeVC can play with this data to identify the repositories that are ahead or behind of other repositories, showing the commits that are missing or that belong to specific branches. We can define a project *p* as a tuple $(R, C, C^{database})$, where R is the set of all cloned repositories of p monitored by DyeVC, C is the set of all commits of p, and $C^{database}$ C is the set of commits of p in the DyeVC database. Each repository $r_i \in R$ is a tuple $\left(R_i^{push}, R_i^{pull}, C_i^{previous}, C_i^{current}, B_i\right)$, where $R_i^{push} \subseteq R$ is the set of repositories that r_i is allowed to push to, $R_i^{pull} \subseteq R$ is the set of repositories that r_i is allowed to pull from, $C_i^{previous} \subseteq C$ is the set of commits in r_i in the previous execution of DyeVC, $C_i^{current} \subseteq C$ is the set of commits in r_i in the current execution of DyeVC, and B_i is the set of named branches of r_i (Fig. 14a). It is worth noting that, as C is the set of all commits of project p (i.e., the domain of commits of p), any set of commits that belong to specific repositories $r_i \in R$, such as $C_i^{previous}$ and $C_i^{current}$, should also belong to C.

Each commit $c_j \in C$ has a set of parent commits $C_j^{parent} \subset C$. Commits are organized in a directed acyclic graph (Fig. 14b), where the first commit of the project has no parent (e.g., commit A in Fig. 14b), revision commits have only one parent (e.g., commit B in Fig. 14b), and merge commits have two or more parents (e.g., commit I in Fig. 14b). All reachable commits from c_j form its history, including c_j itself and the transitive closure over its parents (e.g., {A, B, E, F, H, I, J} is the history of commit J in Fig. 14b). The history of $c_j \in C$ is formally defined as:

$$H_j = \left\{ c \in C | c = c_j \vee \exists c_k : \left(c_k \in C_j^{parent} \wedge c \in H_k \right) \right\}$$

At this point, it is important to notice that ordering is not important for accounting which commits belong to each repository. The only situation in which ordering is important is when DyeVC plots the commit history graph. In this case, DyeVC accesses the tip of the branches (fast operation, as each branch has a reference to its tip) and traverses its transitive closure for plotting all previous commits (also fast, because each commit has a reference to its parents). Note that the commit graph is a directed acyclic graph (DAG), and this DAG is already represented in terms of pointers in C.

The sets of previous and current commits in a repository r_i are updated periodically, according to the monitoring frequency parameter defined by the DyeVC user. In the first execution of DyeVC over r_i, $C_i^{previous} = \varnothing$ and $C_i^{current}$ is populated with all commits obtained directly from Git. In the following executions, $C_i^{previous}$ is populated with the commits *in* $C_i^{current}$ of the previous execution and $C_i^{current}$ is again populated with all commits obtained directly from Git.

Each branch $b_k \in B_i$ is a tuple $(name, c_k)$, where $name$ is the name of b_k and $c_k \in C$ is the tip (i.e., head) of b_k. Consequently, H_k C contains all reachable commits of b_k.

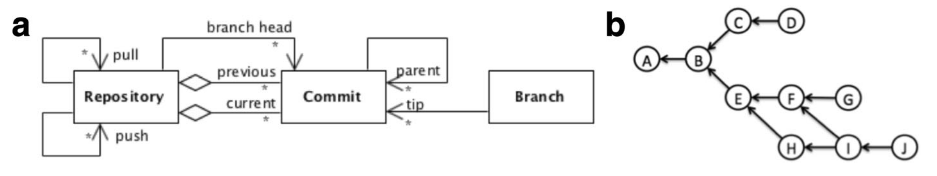


Fig. 14 UML class diagram representing the DyeVC formalization (**a**) and a directed acyclic graph of commits (**b**)

With this foundation established, we can now formalize the process of updating commits in the topology. For a local repository $r_i \in R$ being monitored by DyeVC, the rare situations where a commit is deleted can be formally defined as:

$$Del_i = C_i^{previous} \backslash C_i^{current}$$

Each locally deleted commit $c \in Del_i$ should be removed from $C^{database}$ if no other repository $r \in R$ still contains this commit. Conversely, the new commits in $r_i \in R$ since the previous monitoring cycle can be formally defined as:

$$New_i = C_i^{current} \backslash C_i^{previous}$$

Each locally added commit that is not already in the database ($c \in New_i \setminus C^{database}$) should be inserted in $C^{database}$. This verification is necessary because some of the locally added commits might have already been inserted into the database by another instance of DyeVC.

Moreover, we can formalize the identification of repositories that contain a specific commit and the repositories that are ahead or behind of a given repository. This information is necessary for building some of our visualizations. We formally define the repositories that contain a commit $c_j \in C^{database}$ as:

$$R_j = \left\{ r_i \in R \middle| c_j \in C_{\mathrm{i}}^{current} \right\}$$

We formally define from which repositories r_i is ahead or behind as:

$$Ahead_i = \left\{ r_j \in R_i^{push} \middle| \exists c \in C_i^{current} : c \notin C_j^{current} \right\}$$

$$Behind_i = \left\{ r_j \in R_i^{pull} \middle| \exists c \in C_j^{current} : c \notin C_i^{current} \right\}$$

Finally, we can also formalize the commits that are ahead or behind two specific repositories and the branches in which a commit belongs. This relationship among commits and repositories/branches is also necessary for some of our visualizations. Considering two repositories $r_i, r_j \in R$, we formally define the commits ahead or behind r_i regarding r_j as:

$$Ahead_{i,j} = C_i^{current} \; C_j^{current}$$

$$Behind_{i,j} = C_j^{current} \; C_i^{current}$$

Considering a given repository r_i, we formally define the branches that a commit $c_j \in C$ belongs to as:

$$B_{i,j} = \left\{ b_k \in B_i \middle| c_j \in H_k \right\}$$

The computation of R_j, $Ahead_i$ and $Behind_i$ is not expensive. The set of repositories R is usually small (one or few repositories per developer) and the complexity of the operation for checking if a commit belongs to a specific repository is O(1) (i.e., the complexity of checking if an element belongs to a hash-based set). So, we can say that the complexity of obtaining the relationship of commits and repositories is O(n), where n is the number of repositories in the project.

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 (Eclipse Foundation 2014). 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 (Fielding 2000) 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. We implemented a *MongoLab Provider* to use this RESTful API, which translates the application methods into RESTful commands and vice-versa. It also serializes/deserializes the application objects to/from JSON (*JavaScript Object Notation*) representations to be used through the RESTful commands.

A central document database was chosen because this way DyeVC instances can easily send and gather information. MongoDB was the chosen database because it is free, open-source and cross-platform. Besides, it has many features to improve performance and availability, such as document indexing, replication, and load balancing. Furthermore, it provides RESTful APIs, as cited before.

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

To evaluate our approach, we first conducted a *posthoc* evaluation 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 evaluation involving four participants that used DyeVC. This evaluation also used the JQuery project. Finally, we ran DyeVC over some open-source projects of different sizes and from different sources, aiming at evaluating the scalability of our approach.

5.1 Posthoc evaluation

We conducted a *posthoc* evaluation using a real open source project to demonstrate that our approach can help in answering questions Q1-Q3. The selected project, JQuery, began in 2006 and had 6222 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 on 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 on 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 relevant, 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.

Finally, if rebases were conducted at the repository, this *posthoc* evaluation had no means to detect it, once a rebase consists on rewriting the local history for placing parallel commits on top of existing commits, consequently leaving no trail of the parallel work. This missing detail is not important for our evaluation as well, because this operation is done solely with the purpose of cleaning the repository, leaving its history easier to understand. However, other *posthoc* studies that intend to use DyeVC for finding all cases of parallel work should consider rebase as a potential threat to validity.

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 usernames: *jeresig*, *adam*, and *aakosh*. Figure 15 shows the topology view on Sep 24 2010[5] when *aakosh* had 121 commits pending to be pushed to the central repository (hereafter called *central-repo*). Figure 16 shows part of *aakosh's* commit history and how DyeVC represents commits pending to be pushed (green nodes).

Later on, *aakosh* 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 17 shows the topology view after this registration on Sep 27 2010.[6] 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 (Fig. 17) shows all the clones where it is running, and also discovers other clones connected to them, even if it is not running there.

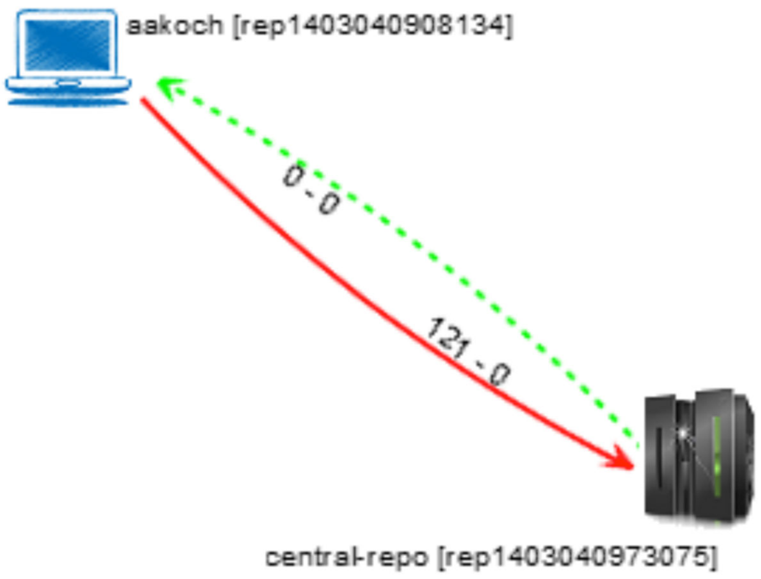


Fig. 15 First monitored repository in Topology view (Sep 24 2010)

Q2: *What are the communication paths among different clones?* DyeVC's topology view (Fig. 17) shows the dependencies between 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 Fig. 18a). 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. Fig. 18b 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 Fig. 17, 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 (Fig. 19) 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 Fig. 18b, we see that the only yellow commit is *a0887*, made by *aakoch*. This tells us that *jeresig* pulled changes from *central-repo* just before *aakoch* pushed commit *a0887*. If we look at Fig. 20, we see that all 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:

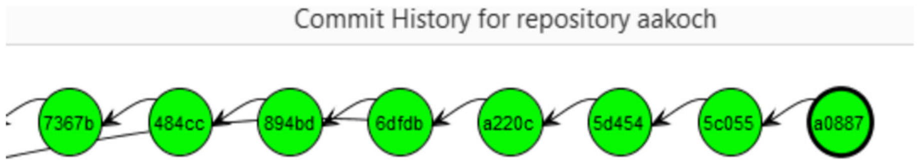


Fig. 16 aakoch's commit history showing commits pending to be pushed

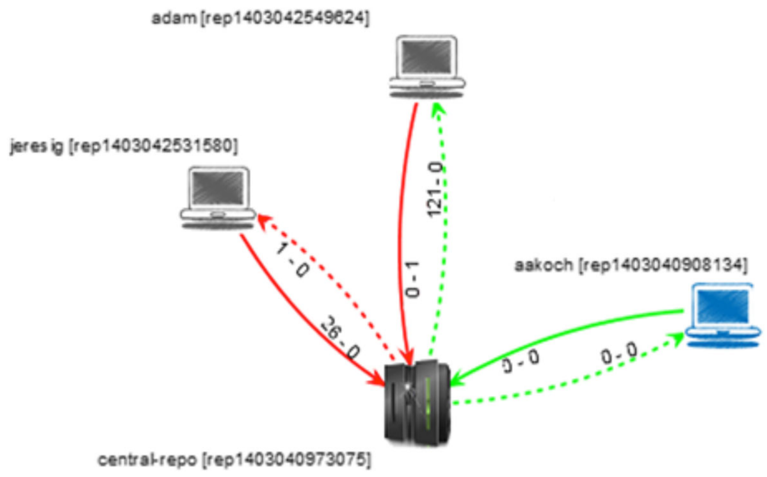
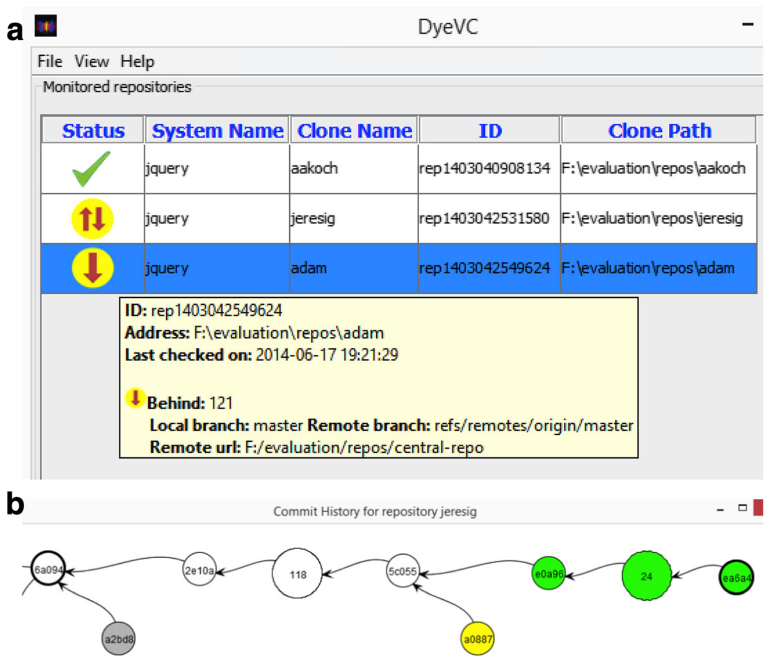

adam [rep1403042549624]
jeresig [rep1403042531580]
aakoch [rep1403040908134]
central-repo [rep1403040973075]
121 - 0
0 - 1
1 - 0
26 - 0
0 - 0
0 - 0

Fig. 17 Three monitored repositories in Topology view (Sep 27 2010)

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 Fig. 18a and Fig. 20). 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 Fig. 16 and Fig. 19) 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).

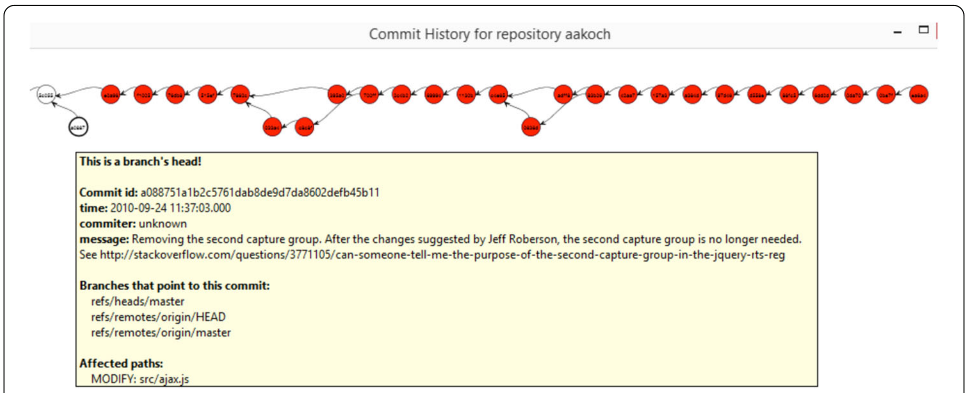


Status	System Name	Clone Name	ID	Clone Path
✓	jquery	aakoch	rep1403040908134	F:\evaluation\repos\aakoch
⇅	jquery	jeresig	rep1403042531580	F:\evaluation\repos\jeresig
↓	jquery	adam	rep1403042549624	F:\evaluation\repos\adam

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

Fig. 18 Adam's tracked branches (**a**) and collapsed commit history for repository jeresig (**b**)



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

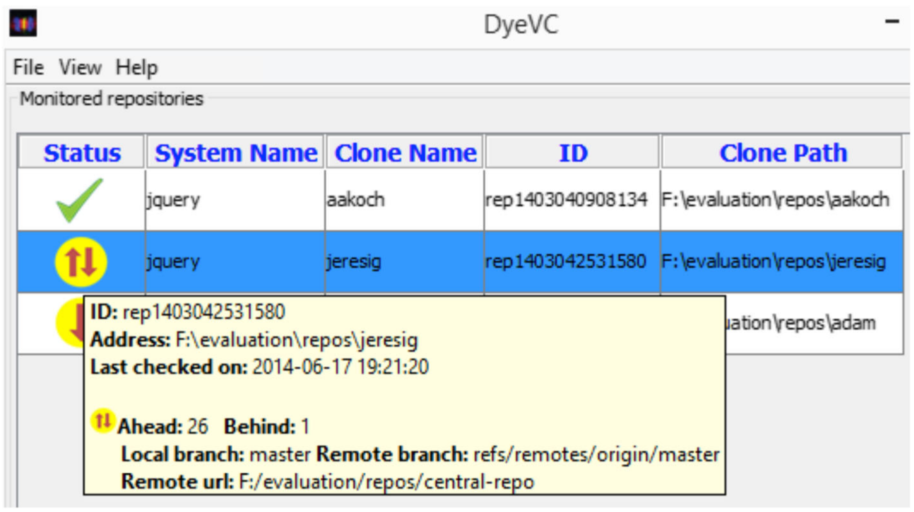
Affected paths:
MODIFY: src/ajax.js

Fig. 19 Aakoch's commit history

5.2 Observational evaluation

We conducted an observational evaluation over the same project used in the *posthoc* evaluation (JQuery) to assess the capability of the visualizations provided by DyeVC in supporting developers and repository administrators. The evaluation 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 goal of this observational evaluation was to analyze when DyeVC helps on understanding the project history better than existing tools. The evaluation was divided into 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



Status	System Name	Clone Name	ID	Clone Path
✓	jquery	aakoch	rep1403040908134	F:\evaluation\repos\aakoch
⇅	jquery	jeresig	rep1403042531580	F:\evaluation\repos\jeresig
				uation\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

Fig. 20 Jeresig's tracked branches

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 evaluation: *gitk***,** *Tortoise Git***,** *Git Bash*, and *SourceTree*. Participants were allowed to access the Internet and search any other procedure or tool that could help in answering the questions. After that, the subject watched a 10-min video presenting DyeVC and started Phase 2 (with DyeVC), which consisted of 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. The values include the time to understand the question, investigate repositories with available tools, look for the answer, and write down the answers in the form. However, the values do not include the time spent filling the consent form and the characterization form, watching the video about DyeVC, and filling the exit questionnaire. 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), none of the subjects managed to answer the questions in Phase 1 (without using DyeVC). For this reason, times shown in Phase 1 are the times spent by the subjects until they gave up finding an answer.

In Phase 1, each subject used different ways to look for the answers. In Phase 2, subjects correctly used DyeVC to find the answers. Question 1.1 was answered using DyeVC Level 3 visualization (Tracked branches). Question 1.3 was answered using Level 4 visualization (Commit History). Finally, questions 1.2 and 2.1 through 2.4 were answered using Level 2 visualization (Topology). Almost all subjects answered all the questions similarly, except for subject P4 in question 1.2 from Phase 1.

Subject P1 answered questions 1.1 and 1.3 in Phase 1 using the command line interface. To answer question 1.1, she looked at the log for both local and remote repositories, counted down how many hashes there were in each log and subtracted these numbers to find the answer. Question 1.3 was answered with *git show* command, which shows, for each affected file in the commit, what has changed. The answer to this question was easy to find because only one file was affected, but if many files had been affected, the subject would have trouble finding all affected files using this procedure. For questions 1.2 and 2.1 through 2.4, the subject tried to find a way to discover related clones by searching the Internet. After a few searches with no promising results, the

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	10	6
P2	13	6	4	5
P3	3	2	2	4
P4	10	2	6	10

subject gave up, and her answer was “I don’t know”. Once there was no answer to question 2.1, next questions in Scenario 2 could not be answered as well.

Subject P2 answered question 1.1 by issuing the *git status* command. To answer question 1.3, she used *Tortoise Git* and walked through the commit tree until finding the desired commit. For questions 1.2 and 2.1 through 2.4, the subject answered that she did not know a way to find an answer. When answering question 2.1, the subject commented that, as a repository manager, she should know which were the existing clones and their relationships, but she did not have any resources available to accomplish that.

Subject P3 answered question 1.1 by issuing a *git status* command (same as subject P2). To answer question 1.3, she used *Tortoise Git* but found the desired commit using the search feature of the tool, instead of walking through the commit tree. For questions 1.2 and 2.1 through 2.4, the subject answered that it was not possible to find an answer.

Subject P4 answered questions 1.1 and 1.3 using *SourceTree*. This subject answered question 1.2 differently from the others. She wrote down each different author of each commit as if it was a different clone. Although this is a valid interpretation, it may happen that authors commit changes in the same clone, and this would lead to a wrong answer for this question. For questions 2.1 through 2.4, the subject answered that it was not possible to find an answer.

The overall results of this evaluation were positive. In Phase 1 (without DyeVC), subjects were able to correctly answer questions Q1.1 and Q1.3 whether using DyeVC or not. Also, further questions were answered correctly only by using DyeVC.

The subjects also answered an exit questionnaire[7] (Cesario 2015). 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. Also, by using Product Reaction Cards, three out of the four subjects stated that DyeVC is helpful and easy to use. Product Reaction Cards (Benedek and Miner 2003) have a large set of words, both positive and negative, used to check the emotional response of a product or design.

5.3 Performance evaluation

We measured the time spent to perform the most common DyeVC operations to evaluate the scalability of our approach. 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 (the number of commits, disk usage, and the 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. Each operation was performed once for each repository, except for the repository registration, which was executed twice (“Insert 1st” and “Insert 2nd”), as detailed below.

We measured the main operations of our approach: “Insert 1st”, invoked when the user registers the first repository of a given system to be monitored; “Insert 2nd”, invoked when the user registers a repository to be monitored in a system that already has

Table 4 Scalability results of DyeVC for repositories with different sizes

Repository	Hosting	Repository metrics			Foreground operations			Background operations times (s)			
		Number commits	Size (MB)	Number files	Commit History		Topology	Insert 1st	Insert 2nd	Check Branches	Update Topology
					Time (s)	Memory Usage[a]	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
JGgit	eclipse.org	2979	10.0	1595	18.4	512	3.4	42.4	46.0	5.9	6.8
EGit	eclipse.org	3775	27.0	1478	21.3	559	3.7	49.6	46.6	4.2	7.3
jQuery	github.com	5518	20.0	253	65.0	1121	4.1	40.0	37.4	1.4	9.4
Tortoise Git	code.google.com	6166	85.0	3220	68.0	492	4.2	39.0	36.0	1.6	9.6
Git Extensions	github.com	6417	448.0	1549	73.0	1529	17.0	155.8	129.0	1.6	10.6
Drupal	drupal.org	23,922	84.4	9290	-	-	18.0	102.0	95.0	2.0	18.0
ExpressoLivre	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	2656	-	-	19.4	196.0	158.6	3.4	40.0

[a]Memory usage was measured in MB during the execution of "Commit History" operation

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 monitored repositories, searching for ahead or behind commits; and "Update Topology", invoked periodically to update 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.

It may be noted 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). The computer used in this evaluation was configured with a 2 GB maximum Java Heap Size, which let us analyze repositories with up to 6 K 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 (Spearman 1904). 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.

However, it is possible to find some more tricky situations, which demonstrate that all three variables (number of commits, size, and number of files) should be taken into consideration when analyzing the performance of each DyeVC operation. One such situation is the one that occurs with *Git Extensions*, which has significantly fewer commits than repositories such as *Git*, but presents times for "Topology", "Insert 1st" and "Insert 2nd" operations in the same level of magnitude. This is because these operations are very I/O intensive. When a repository is registered to be monitored, DyeVC creates the working copy for that repository, as discussed in Section 3.1. Larger repositories will then take more time to perform these actions. Note in Table 4 that the size

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

Operation	Number commits	Size	Number 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
CommitHistory	1.00	0.96	−0.04
Topology	1.00	0.88	0.52

of *Git Extensions* is notably bigger than any other of the repositories used in the evaluation.

Finally, it is worth mentioning that, even with all measurements taken in the same period of the day and from the same machine, short network latencies and processor usage peaks may have occurred, which affect the results.

All in all, although we cannot affirm that DyeVC is scalable to all possible repositories, our evaluation helped us to identify the scalability limits of DyeVC. Without automatic collapses, DyeVC was able to process repositories with around 6500 commits. To put this number in perspective, 99% of 50,012 projects analyzed by Rainer and Gale (2005) had less than 3137 commits. Moreover, Kalliamvakou et al. (2014) indicate that 90% of the projects in GitHub have less than 50 commits. This shows that DyeVC is scalable for a large number of projects, although we still see space for improvements, as presented in the following section.

5.4 Automatic collapsing evaluation

We also studied the impact of the automatic collapsing algorithm in the "Commit History" operation performance. This evaluation was performed at a later time in comparison with the results obtained in the previous section. Consequently, the repository metrics are slightly different. The repository size, number of commits, and number of files are higher, as shown in Table 6.

The design of the evaluation was as follows. First, the "Commit History" operation was performed without using the automatic collapsing. Afterward, sequential and parallel collapse strategies described in Section 3.2.5 were used to simplify the structure of the commit graph, collapsing the corresponding node structures. The execution of the sequential strategy was the first stage, and the parallel strategy was the second stage of each iteration of the automatic collapsing algorithm. Moreover, after each stage, running time and memory consumption were measured. The evaluation was executed in a Core i7 CPU at 2.00 GHz, with 16GB of RAM running Windows 7 64 bits.

We evaluated the capability of the automatic collapsing algorithm to reduce the number of nodes in the commit graph without compromising the quality of the information

Table 6 Characterization of the repositories used in the evaluation of the automatic collapsing algorithm

Repository	Characteristics		
	Size (MB)	Number files	Number commits
DyeVC	3.2	745	228
SAPOS	18.8	668	1245
JGit	39.3	1902	4741
EGit	63.6	1779	4983
jQuery	29.2	296	7291
Git Extensions	94.9	1710	8146
Tortoise Git	168	3518	8442
Drupal	176	10,285	38,047
ExpressoLivre	366	21,592	27,079
Git	104	3026	46,794

shown in the graph (collapses are performed just for commits of the same type, i.e., same color, as discussed in section 3.2.4).

Table 7 shows the reduction achieved after two iterations of the algorithm. The number of iterations was set to two due to empirical observation that there was almost no reduction after a second iteration. With two iterations, the algorithm can reduce the number of nodes by an average of 73% compared to the original graph. In some cases, such as Drupal or ExpressoLivre, which are repositories that we could not analyze before, the nodes reduction surpassed 90%, allowing us to visualize their commit history graph after the automatic collapsing process.

Furthermore, we analyzed the running time and memory consumption of the "Commit History" operation. In particular, data collected for repositories that were visualized before and after collapse are represented in Fig. 21 and Fig. 22 using boxplots. The figures show that the more collapse stages we execute, the less time is needed to represent the commit history, and the less memory is consumed for this purpose. This can be explained by the fact that the automatic collapsing algorithm is linear and very fast comparing to the subsequent visualization process, and speeds up the presentation of the commit graph. Using this method, significantly lower running times and memory consumption values are obtained, compared with values before the automatic collapsing. It was possible to visualize repositories with tens of thousands of nodes (Drupal and ExpressoLivre), which could not be represented before, without applying automatic collapsing process.

Git was the only repository that was not represented visually, even after the automatic collapsing. The main contributing reasons for this fact are its high number of nodes, its low nodes reduction rates, and its inherent complexity.

In the case of Drupal and ExpressoLivre repositories, high nodes reduction rates seem to be influenced by a somewhat more linear structure of the commit graph. There are long chains of 2-degree nodes, corresponding to sequential work stages performed by one contributor. Instead, Git repository showed resistant to collapse. To explain what we mean by "intrinsic complexity" of Git, we identified some structures that prevent commit graph's reduction. An example is shown in Fig. 23. Given the current definition of the collapse operations, the whole structure cannot be reduced because the

Table 7 Reduction of the number of nodes by the automatic collapsing algorithm

Repository	Before Collapse	Iteration 1			Iteration 2		
		1st stage	2nd stage	Reduction (%)	1st stage	2nd stage	Reduction (%)
DyeVC	228	73	47	67.98	32	32	85.96
SAPOS	1245	456	404	63.37	378	375	69.88
JGit	4741	3015	2751	36.41	2635	2635	44.42
EGit	4983	3007	2564	39.65	2347	2329	53.26
jQuery	7291	867	709	88.11	609	603	91.73
Git Extensions	8146	4083	3833	49.88	3702	3684	54.78
Tortoise Git	8442	1466	945	82.63	497	482	94.29
Drupal	38,047	903	697	97.63	563	557	98.54
ExpressoLivre	27,079	3008	2792	88.89	2669	2669	90.14
Git	46,794	24,459	24,216	47.73	24,094	24,094	48.51
Average				66.23			73.15

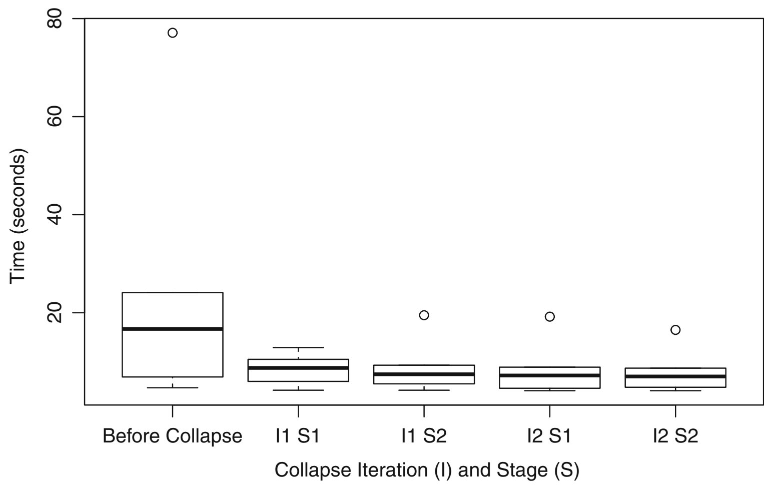


Fig. 21 Boxplot showing the running time of "Commit History" operation depending on the number of executed collapse stages

branches are not sequential chains of commits. Additional automatic collapsing heuristics that consider the possible dependencies between different branches seem necessary to accommodate these cases.

5.5 Threats to validity

While we have taken care to minimize threats to the validity of the evaluations, some factors can influence the results. The usage of a *posthoc* evaluation to assess 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

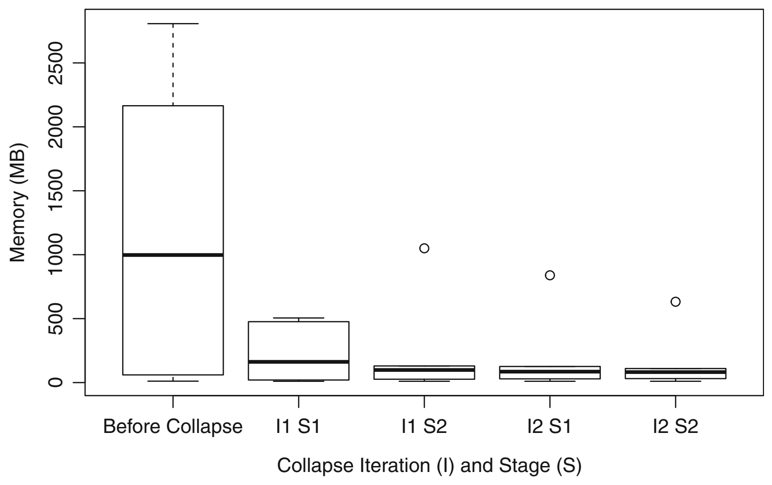


Fig. 22 Boxplot showing the memory consumption of "Commit History" operation depending on the number of executed collapse stages

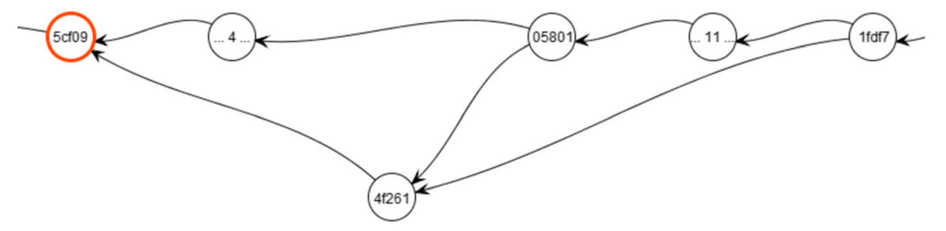


Fig. 23 Example of a structure that prevents automatic collapsing

standpoint. Furthermore, we used an open source project to perform the *posthoc* evaluation, but the *modus operandi* of peers may be different in academic or industrial contexts.

In the observation evaluation, the selection of subjects was made 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 evaluation. 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. Also, 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 (2007), software visualization can be separated into three aspects: structure, behavior, and evolution. DyeVC relates primarily to the evolution aspect, more specifically with studies that aim at improving the awareness of developers that work with distributed software development. Steinmacher et al. (2012) present 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 ("*Commit notification*") includes approaches that notify commit activities. The second group ("*Awareness of concurrent changes*") comprises approaches that not only give the developer awareness of concurrent changes but also inform them about conflicts. The third group ("*Repository visualization*") includes approaches that visualize repository information. Finally, the fourth group ("*DVCS clients*") contains commercial and open source DVCS clients.

The first group contains tools such as SVNNotifier,[8] SCMNotifier,[9] Commit Monitor,[10] SVN Radar,[11] Hg Commit Monitor[12] and Elvin (Fitzpatrick et al. 2006). 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 on 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 (Sarma and van der Hoek 2002), CollabVS (Dewan and Hegde 2007), Crystal (Brun et al. 2011), Lighthouse (da Silva et al. 2006), FASTDash (Biehl et al. 2007), and WeCode (Guimarães and Silva 2012). 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 previously 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 (Collberg et al. 2003), classes (Lanza 2001), lines (Voinea et al. 2005), authors (Gilbert and Karahalios 2006), and branch history (Elsen 2013)[13,.14] 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,*[15]*Tortoise Git,*[16] *EGit for Eclipse,*[17] and *SourceTree.*[18] The data about commits shown by these tools varies, but usually 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.

Table 8 compares DyeVC with each group used to classify related work presented in this section, according to the following features: notifications (Which types of notification the approach supports?); CVCS (Does the approach support CVCS?); DVCS (Does the approach support DVCS?); related repositories (Does the approach identifies related repositories?); levels of detail (Does the approach present information in different levels of detail?); multiple peers (Does the approach support repositories with multiple peers, i.e., multiple pull / push destinations?); commits in peer nodes (Does the approach detects commits in peer nodes, i.e., nodes that have a direct communication path to each other?); commits in non-peer nodes (Does the approach detect commits in non-peer nodes, i.e., nodes that do not have a direct communication path to each other?); multiple branches (Does the approach support multiple branches in DVCS?); topology (Does the approach supply any topology visualization that shows dependencies among repositories?); and, finally, commit History (Does the approach allow visualizing only a partial commit history, showing only local commits, or does it allow visualizing a full commit history, including commits in other repositories that were not synchronized yet, or that are in non-tracked branches?).

Table 8 Comparing DyeVC features with related work

Feature	Commit notification	Awareness of concurrent changes	Repository visualization	DVCS clients	DyeVC
Notifications	New commits	Conflicts	No	No	Status change against peers
CVCS	Yes	Yes	Yes	No	No
DVCS	Some[a]	Some[b]	Some[c]	Yes	Yes
Related repositories	No	No	No	No	Yes
Levels of detail	No	No	No	No	Yes
Multiple peers	No	No	No	No	Yes
Commits in peer nodes	No	Some[d]	Some[e]	No	Yes
Commits in non-peer nodes	No	No	No	No	Yes
Multiple branches	No	No	No	Yes	Yes
Topology	No	No	No	No	Yes
Commit history	No	No	Some[f] / Partial[g]	Partial[g]	Full

[a]Exceptions are SCM Notifier and Hg Commit Monitor
[b]Exception is Crystal
[c]Exceptions are VisGi, Visugit, and GitHub's Network Graph
[d]Exception is Lighthouse
[e]Exception is GitHub's Network Graph
[f]Visugit and GitHub's Network Graph
[g]Approaches allow visualizing only local commits. Commits in other repositories that were not synchronized yet, or that are in non-tracked branches, are not shown

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, at 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 provide awareness and safety better when working with DVCS.

6.1 Trace reduction methods and automatic collapsing

Trace reduction is the compression of traces in some manner (either lossless or lossy) so that they can be stored and processed efficiently (Kaplan et al. 1999; Mohror and Karavanic 2009). The process of collapsing the commit graph can be seen as a particular case of trace reduction.

Program analysis and software visualization communities have already proposed trace reduction methods (Kuhn and Greevy 2006; Cornelissen et al. 2008; Noda et al. 2012; Jayaraman et al. 2017). In (Kuhn and Greevy 2006) and (Cornelissen et al. 2008), a trace reduction technique is mentioned, which assigns consecutive events that have equal or increasing nesting levels to the same group. Particularly, method call sequences are summarized in one method-call chain (Kuhn and Greevy 2006).

Also, compact sequence diagram generation is studied in (Noda et al. 2012) and (Jayaraman et al. 2017). To have a better sequence diagram representation of the program execution, Noda's method (Noda et al. 2012) abstracts the history of object interaction by grouping strongly correlated objects. These objects are compacted, achieving an appropriate reduction in the number of objects appearing in the sequence diagram, which results in compression of this diagram along the horizontal axis. Also, (Jayaraman et al. 2017) presents vertical and horizontal sequence diagram compaction techniques. For this end, one-to-one correspondence between call trees and sequence diagrams is used. A maximally compacted tree is obtained, generating smaller and more useful diagrams.

These approaches work with method-call sequences and sequence diagrams, both having a tree structure. Unlike these works, when applying automatic collapsing, we deal with a commit graph, which is a DAG, with a high number of commit nodes. The structure of a graph is more rich and complex than the structure of a tree.

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 evaluation 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.

Some future research topics arise from this work. 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. Moreover, the formalization of DyeVC mechanics could be used to prove correctness properties of our implementation. Finally, some optimization should be done to allow DyeVC work with larger repositories with more complex branch structures.

8 Endnotes

[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.

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

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

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

[5]Considering the scenario just after commit *a088751a1b2c5761dab8de9d7da8602def b45b11.*

[6]Considering the scenario just after commit *ea6a4813b7d996f6f7af0b61a5f1bf4ab80b 291d.*

[7]The exit questionnaire can be found in Appendix G of the referenced Master's Thesis, which can be found at https://github.com/gems-uff/dyevc/blob/master/docs/dissertation.pdf.

[8]http://svnnotifier.tigris.org/ (2012)

[9]https://github.com/pocorall/scm-notifier (2012)

[10]http://tools.tortoisesvn.net/CommitMonitor.html (2013)

[11]http://code.google.com/p/svnradar/ (2011)

[12]https://bitbucket.org/dun3/hgcommitmonitor (2009)

[13]Visugit: https://github.com/hozumi/visugit

[14]GitHub's Network Graph: https://github.com/blog/39-say-hello-to-the-network-graph-visualizer

[15]http://git-scm.com/docs/gitk

[16]https://tortoisegit.org/

[17]http://eclipse.org/egit/

[18]http://www.sourcetreeapp.com/

Abbreviations
API: Application programming interface; CM: Configuration management; CPU: Central processing unit; CVCS: Centralized version control systems; DAG: Directed acyclic graph; DVCS: Distributed version control systems; HTTP: Hypertext transfer protocol; HTTPS: HTTP secure; JSON: JavaScript object notation; JUNG: Java Universal network/graph; MB: Megabyte; RAM: Random access memory; RESTful: Representational State transfer; UML: Unified modeling language; VCS: Version control systems

Acknowledgements
We thank CNPq and FAPERJ for the financial support.

Funding
CNPq and FAPERJ sponsored this work.

Availability of data and materials
The source code and the link to download DyeVC via Java Web Start can be found at https://github.com/gems-uff/dyevc. All projects used in the evaluations are available in their respective repositories, described in Table 4.

Authors' contributions
CC contributed in the design and implementation of DyeVC and the design of the automatic collapsing algorithm and was responsible for running the posthoc, observational, and performance evaluations. RI contributed in the design and implementation of the automatic collapsing algorithm and was responsible for running the automatic collapsing evaluation. LM contributed in the design of DyeVC, the automatic collapsing algorithm, and the evaluations, and was responsible for the DyeVC formalization. All three authors contributed to writing the paper. All authors read and approved the final manuscript.

Competing interests
The authors declare that they have no competing interests.

9 Publisher's Note

Received: 9 January 2017 Accepted: 14 July 2017
Published online: 21 July 2017

References

Appleton B, Berczuk S, Cabrera R, Orenstein R (1998) Streamed lines: branching patterns for parallel software development, Pattern languages of programs conference (PLoP), p 98

Benedek J, Miner T (2003) Measuring desirability: new methods for evaluating desirability in a usability lab setting. In: Proceedings of usability professionals association (UPA), Orlando, p 57

Biehl JT, Czerwinski M, Smith G, Robertson GG (2007) FASTDash: a visual dashboard for fostering awareness in software teams. In: ACM conference on human factors in computing systems (CHI). ACM, San Jose, pp 1313–1322

Brun Y, Holmes R, Ernst MD, Notkin D (2011) Proactive detection of collaboration conflicts. In: ACM SIGSOFT symposium and European conference on foundations of software engineering (ESEC/FSE). ACM, Szeged, pp 168–178

Cederqvist P (2005) Version management with CVS. Free Software Foundation

Cesario C (2015) Awareness over distributed version control systems. Master's thesis, Universidade Federal Fluminense - UFF

Cesario CM, Murta LGP (2016) Topology awareness for distributed version control systems. In: Proceedings of the 30th Brazilian symposium on software engineering (SBES). ACM, Maringá, pp 143–152

Chacon S (2009) Pro Git, 1st edn. Apress, Berkeley

Collberg C, Kobourov S, Nagra J, Pitts J, Wampler K (2003) A system for graph-based visualization of the evolution of software. In: ACM symposium on software visualization (SOFTVIS). ACM, San Diego, pp 77–ff

Collins-Sussman B, Fitzpatrick BW, Pilato CM (2011) Version Control with Subversion. Compiled from r4849. O'Reilly Media, Stanford

Cornelissen B, Moonen L, Zaidman A (2008) An assessment methodology for trace reduction techniques. IEEE International Conference on Software Maintenance, In, pp 107–116

Dewan P, Hegde R (2007) Semi-synchronous conflict detection and resolution in asynchronous software development. In: European conference on computer-supported cooperative work (ECSCW). Springer London, Limerick, pp 159–178

Diehl S (2007) Software visualization: visualizing the structure, behaviour, and evolution of software. Springer, Berlin, New York

Eclipse Foundation (2014) 2014 annual eclipse community report. Eclipse Foundation, San Francisco

Elsen S (2013) VisGi: Visualizing Git branches. In: IEEE working conference on software visualization (VISSOFT). IEEE, Eindhoven, pp 1–4

Estublier J (2000) Software configuration management: a roadmap. In: International conference on software engineering (ICSE). ACM, Limerick, pp 279–289

Fielding RT (2000) Architectural styles and the Design of Network-Based Software Architectures. Thesis, University of California

Fitzpatrick G, Marshall P, Phillips A (2006) CVS integration with notification and chat: lightweight software team collaboration. In: ACM conference on computer-supported cooperative work (CSCW). ACM, Banff, pp 49–58

Gilbert E, Karahalios K (2006) LifeSource: two CVS visualizations. In: ACM conference on human factors in computing systems (CHI). ACM, Montreal, pp 791–796

Guimarães ML, Silva AR (2012) Improving early detection of software merge conflicts. In: Internation conference on software engineering (ICSE). IEEE Press, Zürich, pp 342–352

Gumm D-C (2006) Distribution dimensions in software development projects: a taxonomy. IEEE Softw 23:45–51

Jayaraman S, Jayaraman B, Lessa D (2017) Compact visualization of java program execution. Softw Pract Exp 47:163–191. doi:10.1002/spe.2411

Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and perils of mining GitHub. In: Proceedings of the 11th working conference on mining software repositories. ACM, New York, pp 92–101

Kaplan SF, Smaragdakis Y, Wilson PR (1999) Trace reduction for virtual memory simulations. In: Proceedings of the 1999 ACM SIGMETRICS international conference on measurement and Modeling of computer systems. ACM, New York, pp 47–58

Kuhn A, Greevy O (2006) Exploiting the analogy between traces and signal processing. In: 22nd IEEE international conference on software maintenance, pp 320–329

Lanza M (2001) The evolution matrix: recovering software evolution using software visualization techniques. In: International workshop on principles of software evolution (IWPSE). ACM, Tokyo, pp 37–42

Mohror K, Karavanic KL (2009) Evaluating similarity-based trace reduction techniques for scalable performance analysis. In: Proceedings of the conference on high performance computing networking, storage and analysis. ACM, New York, pp 55:1–55:12

Noda K, Kobayashi T, Agusa K (2012) Execution trace abstraction based on meta patterns usage. In: 19th working conference on reverse engineering, pp 167–176

O'Sullivan B (2009a) Mercurial: the definitive guide, 1st edn. O'Reilly Media, Sebastopol

O'Sullivan B (2009b) Making sense of revision-control systems. CACM 52:56–62

Perry DE, Siy HP, Votta LG (1998) Parallel changes in large scale software development: an observational case study. In: International conference on software engineering (ICSE). IEEE Computer Society, Kyoto, pp 251–260

Rainer A, Gale S (2005) Evaluating the quality and quantity of data on open source software projects. Proceedings of the 1st international conference on open source software

Rochkind MJ (1975) The source code control system. IEEE Trans Softw Eng 1:364–470

Sarma A, van der Hoek A (2002) Palantir: coordinating distributed workspaces. In: 26th computer software and applications conference (COMPSAC). IEEE, Oxford, pp 1093–1097

da Silva IA, Chen PH, Van der Westhuizen C, Ripley RM, van der Hoek A (2006) Lighthouse: coordination through emerging design. In: Workshop on eclipse technology eXchange (ETX). ACM, Portland, pp 11–15

Spearman C (1904) The proof and measurement of association between two things. Am J Psychol 15:72–101. doi:10.2307/1412159

Steinmacher I, Chaves A, Gerosa M (2012) Awareness support in distributed software development: a systematic review and mapping of the literature. In: ACM conference on computer-supported cooperative work (CSCW). ACM, Seattle, pp 1–46

Tichy W (1985) RCS: a system for version control. Soft Pract Exp 15:637–654

Voinea L, Telea A, van Wijk JJ (2005) CVSscan: visualization of code evolution. In: ACM symposium on software visualization (SOFTVIS). ACM, Saint Louis, pp 47–56

Walrad C, Strom D (2002) The importance of branching models in SCM. IEEE Comput 35:31–38

Wloka J, Ryder B, Tip F, Ren X (2009) Safe-commit analysis to facilitate team software development. In: International conference on software engineering (ICSE). IEEE Computer Society, Vancouver, pp 507–517