[[1]](#footnote-1)

DyeVC

Cristiano M. Cesario and Leonardo G. P. Murta

Institute of Computing

Universidade Federal Fluminense, Niteroi, Brazil

{ccesario, leomurta}@ic.uff.br

*Abstract*—Software development using distributed version control systems has become more frequent recently. Such systems bring more flexibility, but they also bring greater complexity to administer and monitor the multiple existing repositories as well as the proliferation of several branches. In this paper we propose DyeVC, an extensible tool to assist developers and repository administrators in identifying dependencies among the distributed repositories in order to help to understand what is going on around one’s repository and depict the relations between the existing repositories.

*Index Terms*—Configuration management, Context awareness, Distributed version control

# INTRODUCTION

V

ERSION Control Systems (VCS) date back to the 70s, when SCCS emerged [1]. Their primary purpose is to keep software development under control [2]. Along these almost 40 years, VCSs evolved from a centralized repository with local access, as in SCCS and RCS [3], to a client-server approach, as in CVS [4] and Subversion [5]. More recently, distributed VCSs (DVCS) arose, allowing clones of the entire repository in different locations, as in Git [6] and Mercurial [7]. According to a survey conducted among the Eclipse community [8], Git and Github combined usage increased from 6.8% to 36.3% between 2010 and 2013 (a growth greater than 600%). During this same period, Subversion and CVS combined usage decreased from 71% in 2010 to 42.3% in 2013. This clearly shows momentum and a strong tendency in the adoption of DVCSs among the open source community.

According with Walrad and Strom [9], creating branches is essential to software development, because it enables concurrent development, allowing the maintenance of different versions of a system, the customization to different platforms and to different customers, among other features that are expected from current software systems. DVCSs include better support to work with branches [10], 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, a commit is issued based on an old revision, or updates are pushed to or pulled from other repositories. All these branches, whether explicit or not, eventually will be reintegrated to their origin by means of merge operations, reflecting the changes made.

Distributed software development, especially from the geographical perspective [11], brings a set of risk factors, configuration management being one of them [12]. The increasing growth of development teams, and their distribution along distant locations – even different continents –, together combined together with the proliferation of branches, introduce additional complexity to notice actions performed in parallel by different developers. According to Perry *et al* [13], concurrent development increases the number of defects in software. Besides, da Silva *et al* [14] say that development tools control concurrent development with the assistance of VCSs, which work in majority with the concept of each developer having a private workspace. This postpones the perception of conflicts that result from changes made by co-workers. These conflicts are noticed only after a pull or a push in the context of DVCS. Moreover, Brun *et al* [15] shows 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 exposed by Appleton *et al* [16], 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 repository. This scenario generates traffic similar to that of peer-to-peer applications. With this diversity of topologies, managing the evolution of a complex system becomes a tough task, making it difficult to find an answer to questions like:

* Which clones were created from a repository?
* What are the dependencies between different clones?
* Which changes are being worked on in parallel (in different clones or different branches)?
* Which conflicts appeared might appear after merging concurrent changes?

Most of existing works deal with the last two questions, giving to the developers the perception of concurrent changes. Palantir [17], FASTDash [18], Lighthouse [14], CollabVS [19], Safe-Commit [20], Crystal [15], WeCode [21], and Polvo [22] are examples of this kind of work. Among these, the only one that deals with multiple branches is Polvo, establishing metrics that assist in determining the merge effort between branches. However, it has a strict focus in Centralized Version Control Systems (CVCS), which are much less prone to branches if compared to DVCS.

This paper expands the concept of DyeVC[[2]](#footnote-2) presented in a previous work [23], which is a novel visualization infrastructure for DVCS, which gathers information about different clones of a repository and presents them visually to the user. This allows one to perceive how his repository evolved over time and how this evolution compares to the evolution of other repositories in the project. The main goal of DyeVC is three-fold: to increase the developer knowledge of what is going on around his repository and the repositories of his teammates; to enable repository administrators to visualize how the several existing repositories of a system interact with each other; and finally, to establish an extensible platform to present different information and metrics regarding distributed repositories.

The rest of this paper is organized as follows. Section II presents a motivational example to this work. Section III presents the approach used in DyeVC. Section IV shows how this work was evaluated. Section V discusses some related work and Section VI concludes the paper and presents future work.

# Motivational Example

Fig. 1 shows a scenario with some developers, each one having a clone of a repository originally created at Xavier Institute. Xavier Institute acts like a central repository, where code developed by all teams is integrated, tested, and released to production. There is a team working in Xavier Institute, leaded by Professor X, and a remote developer (Storm) that periodically receives updates from the Institute. Outside the Institute, Wolverine leads a remote team located in a different continent, 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 only pull updates from Rogue.

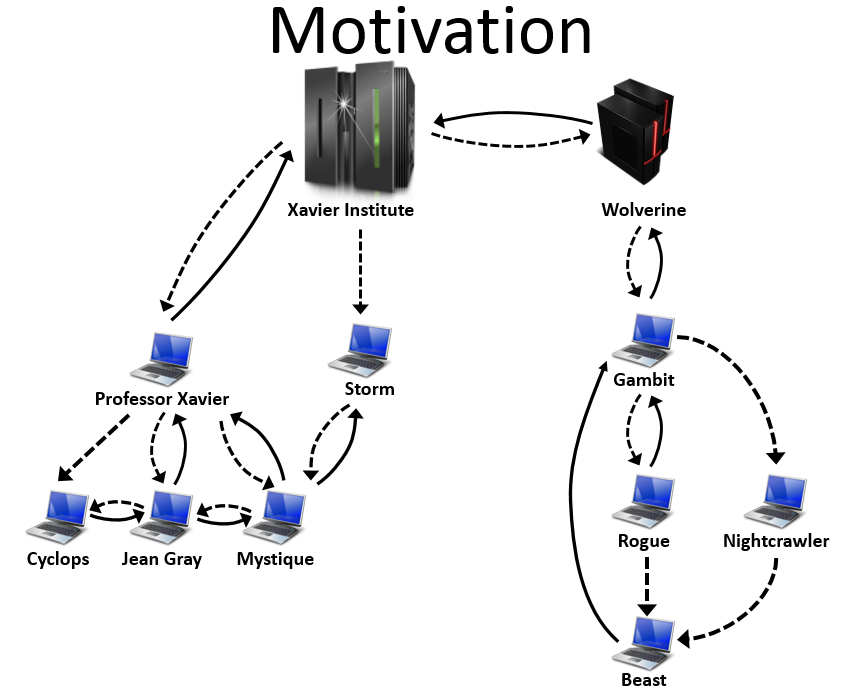


Fig. 1. A development scenario involving some developers

Each one of the developers has a complete copy of the repository and is able to send and receive updates to or from any other developer. Considering the existence of *n* developers, we could reach a total of *n \* (n - 1)* different possibilities of communication. In practice, however, this limit is not reached: while interaction among some developers is frequent, it may happen that others have no idea about the existence of some coworkers, as it occurs with Mystique and Nightcrawler, where there is no direct communication.

As an example, from a developer’s point of view, like Beast, how can he know at a given moment if there are commits in Rogue or in Nightcrawler that were not yet pulled? Alternatively, would be the case that there are local commits pending to be pulled by Gambit? Beast could certainly periodically pull changes from his partners, to check if eventually there were updates, but this would be a manual procedure, prone to be forgotten. What if a tool had the knowledge of Beast’s partners, and constantly monitored those, warning Beast of any local or remote updates that had not been synchronized yet?

From an administrator’s point of view, how can he know what are the existing clones of a system? Who and how do they relate with? How can he know if there are pending commits to be sent from a test repository to a production one?

# Approach

We define DyeVC using the categories proposed by [24]: task, audience, target, medium and representation.

The **tasks** are three-fold. First, DyeVC should work as a non-obtrusive awareness tool to increase the developer knowledge of what is going on around his repository and the repositories of his teammates. Second, DyeVC should enable repository administrators / managers to visualize how the several existing repositories of a system interact with each other. Third, DyeVC should establish an extensible platform to present different information and metrics regarding DVCS.

The **audience** of DyeVC is everyone involved with implementing software through the usage of distributed version control systems. The main roles that DyeVC immediately benefits are system administrators and developers, but due to its extensibility, different views can be developed to support different roles.

The **target** of DyeVC is any local DVCS clone of a repository hosted on a Git server, such as local Git servers, GitHub[[3]](#footnote-3), Gitorious[[4]](#footnote-4) or Bitbucket[[5]](#footnote-5).

The **medium** used by DyeVC can be a simple message in the notification area, or a window in the user’s desktop.

The **representation** should be appropriate to show the information in each different visualization. For instance, the various clones and their relationships use a network topology metaphor and the repository history use a directed acyclic graph visualization.

## Implementation

DyeVC involves continuously monitoring a group of interrelated repositories, starting from repositories registered by the user. The implementation uses Java Web Start[[6]](#footnote-6) Technology, and focus on monitoring Git repositories, once that this is the most used DVCS nowadays [8]. The gathering of information from repositories is accomplished using JGit[[7]](#footnote-7) library, which allows the user to use DyeVC without having a Git client installed. DyeVC presents the information gathered as a series of graphs by using JUNG[[8]](#footnote-8) library, from which it inherits the ability to extend existing layouts and filters to create new ones, which can be dynamically attached to the graphs that it presents.

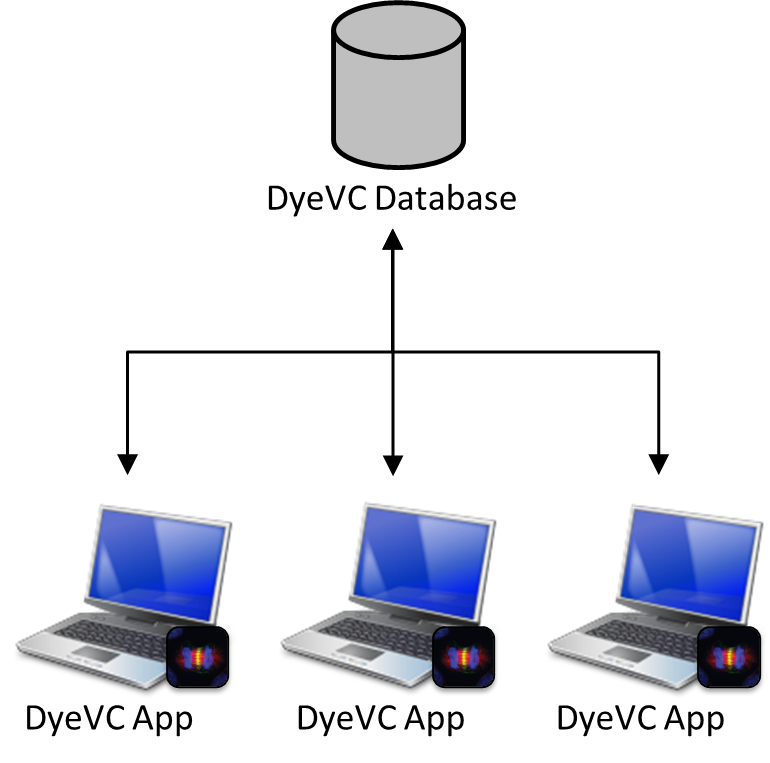


Fig. . How DyeVC gathers information

The information gathered is stored in a central document database running MongoDB[[9]](#footnote-9), using two types of documents:

* The repositories document, comprised of:
  + Repository id;
  + System name, used to group all repositories that are clones of a specific system;
  + Host name of the machine hosting the clone;
  + Clone name, which is given by the user when it registers the clone to be monitored;
  + Clone path in the user’s machine;
  + List of clones to which this repository pushes to;
  + List of clones from which this repository pulls from.
* The commits document, comprised of:
  + Commit id, which is the hash given by Git;
  + Commit date;
  + Committer name;
  + List of repositories where this commit exists;
  + List of this commit’s parents (considering that a commit can be a merge of several parent commits);
  + Commit message;
  + System name.

DyeVC has also the ability to gather information not only from the registered repositories in the user’s machine, but also from its partners, which are the repositories that a given repository communicates.

## Levels of Detail

The information gathered by DyeVC is presented visually in different levels of detail. The application shows notifications whenever there are changes in any of the registered repositories or in its partners, as shown in Fig. 3. The period between subsequent monitor runs is configurable and defaults to 5 minutes.



Fig. . DyeVC showing notifications in the notification area

The levels of detail defined in DyeVC include presenting the status of a given repository against its partners (Level 1); zooming into the branches of the repository, showing the status of each local branch that tracks a remote branch (Level 2); and zooming into the commits of the repository, showing a visual log with information about each commit (Level 3).

Table

Possible Status of a Repository

| Status | Description |
| --- | --- |
| question_32 | DyeVC has not analyzed the repository yet. |
| check_32 | Repository is synchronized with all partners. |
| ahead_ylw_32 | Repository has changes that were not sent yet to its partners (it is ahead its partners). |
| behind_ylw_32 | Partners have changes that were not sent yet to the repository (it is behind its partners). |
| aheadbehind_ylw_32 | Repository is both ahead and behind its partners. |
| nocheck_32 | Invalid repository. This happens when DyeVC cannot access the repository. The reason is presented to the user. |

The status evaluation considers the existing commits in each repository individually. Each commit maps a group of changes in various artifacts and it is uniquely identified in the repository. Moreover, due to the nature of DVCS, where old data is never deleted and commits are cumulative, if a commit N is created over a commit N – 1, the existence of commit N in a given repository implies that commit N – 1 also exists in the repository. Thus, by examining the existence of commits in the local repository not yet replicated to the remote repository, and vice-versa, it is possible to come to one of the situations presented in Table II.

Table

Status of a local repository with regard to a remote one, based on the existence of non-replicated commits in each one of them

| Existence of  non-replicated commits | | Local Status |
| --- | --- | --- |
| Local  Repository | Remote  Repository |
| Yes | Yes | aheadbehind_ylw_32Ahead and Behind (needs push and *pull*) |
| Yes | No | ahead_ylw_32Ahead (needs *push*) |
| No | Yes | behind_ylw_32Behind (needs *pull*) |
| No | No | check_32Synchronized |

To illustrate how this approach works, let us assume that each commit is represented by an integer number. At a giving moment, the local repository of some of the developers from Fig. 1 have the commits shown in Table III.

Table

Existing commits in each repository

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

Considering just the synchronizations presented in Fig. 1, which depend on the direction of the arrows and on the type of the lines, the perception of each developer regarding to his known partners is shown in Table IV. 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.

Table

Status of each repository based on known remote repositories

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Repository | Wolverine | Gambit | Rogue | Nightcrawler | Beast |
| Wolverine | - | - | - | - | - |
| Gambit | check_32 | - | - | - | - |
| Rogue | - | aheadbehind_ylw_32 | - | - | - |
| Nightcrawler | - | ahead_ylw_32 | - | - | - |
| Beast | - | behind_ylw_32 | behind_ylw_32 | behind_ylw_32 | - |

The main window of DyeVC presents Level 1 information, as shown in Fig. 4. Upon a mouse over on a repository, DyeVC informs the number of commits that are ahead or behind in each branch that tracks a remote branch (Level 2 information).



Fig. . DyeVC Main Screen

Level 3 information presents a visual history of the repository (Fig. 5). Each vertex in the graph represents a known commit for the same system, which is named after its hash’s five initial characters. A heavier stroke denotes that the commit is a branch’s head (e.g. commit f1a48). Each commit is painted according to its existence in the local repository and its partners, as following:

* If commit exists locally and in all partners, it is painted in white;
* If commit exists locally but does not exist in any partner it pushes to, it is painted in green;
* If commit doesn’t exist locally, but exists in any of the partners it pulls from, it is painted in yellow;
* If commit exists in repository that is not a partner (this is, it can’t be pulled from it), it is painted in red;
* If commit does not belong to a tracked branch, it is painted in gray.



Fig. 5. Log Window (commit history)

## 

Commits are drawn according to its precedence order. Thus, if a commit N is created over a commit N – 1, then commit N will be located in the right hand side of commit N – 1. DyeVC presents a tooltip with some information about each commit, upon a mouse over a node.

The history window can also be zoomed in or out, whether the user wants to see details of a particular area of the history or an overview of the entire history. The line style can be one of cubic curves, straight or quad curves. By changing the window mode from *transforming* to *picking*, it is possible to select a group of nodes and collapse them into one node that represents them, or simply drag them into new positions to have a better understanding of an area where there are too many crossing lines.

## Topology View

DyeVC presents a topology view showing all repositories for a given system, as depicted in Fig. 6, where each node represents a known clone of the system. Each node icon has a different meaning:

* A blue computer represents your clone;
* Black computers represent ordinary clones;
* Servers represent central repositories that do not pull from nor push to any other clone, or clones where DyeVC is not running;
* Green checkmarks represent selected nodes.

Each edge in the graph represents a relationship between two repositories. Edges with a continuous stroke mean that the source clone pushes to the destination clone. Edges with a dotted stroke mean that the destination clone pulls from the source clone. The edge labels show how many commits from the source clone are missing in the destination clone. If both clones are synchronized, than the edge connecting them is green, otherwise it is red.

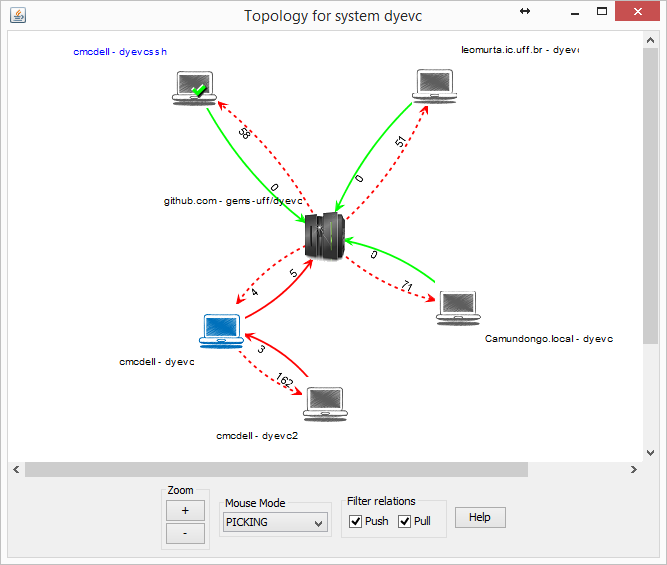


Fig. . Topology view for a given system

# Evaluation

We performed both quantitative and qualitative evaluations of DyeVC, as detailed in the next sub-sections.

## Quantitative Evaluation

In order to evaluate the time spent to perform the most common DyeVC operations, we analyzed repositories of different sizes and hosted in different git servers. The results are in Table V.

Table

Time (in seconds) spent by DyeVC in several operations for repositories with different histories and sizes

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Repository | Hosting | Repository size | | | | Foreground operations | | Background operations | | | |
| # commits | MB | # files | #nodes | Commit History | Topology | Insert 1st | Insert 2nd | Check Branches | Update Topology |
| DyeVC | github.com | 187 | 1 | 539 | 6 | 3,5 | 2,7 | 12,4 | 16,1 | 1,7 | 4,4 |
| Sapos | github.com | 702 | 7 | 685 | 11 | 5,6 | 3,2 | 20,8 | 22,6 | 1,8 | 5,2 |
| jgit | eclipse.org | 2979 | 10 | 1595 | 3 | 18,4 | 3,4 | 42,4 | 46 | 5,9 | 6,8 |
| egit | eclipse.org | 3775 | 27 | 1478 | 3 | 3,7 | 21,3 | 49,6 | 46,6 | 4,2 | 7,3 |
| jquery | github.com | 5518 | 20 | 253 | 3 | 65 | 4,1 | 40 | 37,4 | 1,4 | 9,4 |
| Tortoise Git | code.google.com | 6166 | 85 | 3220 | 3 | 68 | 4,2 | 39 | 36 | 1,6 | 9,6 |
| Gitextensions | github.com | 6417 | 448 | 1549 | 3 |  | 17 | 155,8 | 129 | 1,6 | 10,6 |
| drupal | drupal.org | 23922 | 84,4 | 9290 | 3 |  | 18 | 102 | 95 | 2 | 18 |
| Expresso Livre | gitorious.org | 25822 | 141 | 20729 | 3 |  | 18,2 | 110 | 102 | 2,1 | 19,3 |
| Git | github.com | 35260 | 98 | 2656 | 3 |  | 19,4 | 196 | 158,6 | 3,4 | 40 |

Table V shows the monitored project (name and hosting service), the repository size – in terms of number of commits, disk usage, number of files and number of nodes in the measured topology –, and the time spent by DyeVC to run some background and foreground operations. All measurements were done in the same period of the day and from the same machine, a Core Duo CPU running at 2.53 GHz, with 4GB RAM running Windows 8.1 Professional 64 bits, connected to the internet at 35 Mbit/s. We measured the following operations:

* Insert 1st: When the user includes the first repository of a given system to be monitored;
* Insert 2nd: When the user includes a repository to be monitored in a system which already have repositories registered;
* Commit History: When the user selects the command to show the commit history of a given repository;
* Topology: When the user selects the command to show the topology of repositories of a given system;
* Check Branches: The monitor runs this process periodically to check all the monitored repositories, searching for ahead or behind commits;
* Update Topology: The monitor runs this process periodically to update the topology information in MongoDB. It updates the existing repositories, their partners and the existing commits, marking in which repositories each commit is found.

Table V shows that, despite of the other size variables, the number of commits in the repositories made the difference in the measurements. We had longer operations in repositories that had more commits. The slowest operations were Insert 1st and Insert 2nd, due to the amount of data sent over the internet to update the database. Among the foreground operations, the “Topology” operation, which shows information in a low level of granularity, had a significant increase in its response time, but with lower values than the “Commit History” operation, which deals with much more granular data, showing information about each one of the commits in the topology. In fact, the application was not able to show the commit history for repositories with more than 6.4K commits, giving memory errors. As the commit graph has to be entirely in memory to be plotted, this is a scalability limitation of our approach.

## Qualitative Evaluation

To be written

# Related Work

According to [25] software visualization can be separated into three aspects: structure, behavior and evolution. DyeVC relates primarily with the evolution aspect, more specifically with studies that aim at improving the perception of developers that work with distributed software development (awareness tools). Awareness is defined by [26] as “an understanding of the activities of others to provide a context for one’s own activities”. A recent work by [27] presents a systematic review of such studies and classify them according to the Gutwin et al. Awareness Framework elements [28]. The classification is not exclusive, i.e., a given tool can present several elements of the Awareness Framework. Following this framework, DyeVC can be classified as a Workspace Awareness tool, presenting the following awareness elements:

* and which repositories interact one with another;
* Location: DyeVC associates a hostname with commits and repositories;
* Actions: DyeVC shows a commit history that identifies the person responsible for that commit and the actions executed, at the file level;
* Extents: DyeVC identifies in the commit history the commits that one has, those that can be obtained and those that are not accessible.
* Sphere of Influence: In the topology view, DyeVC identifies which repositories are accessible from a given one, and allows seeing who can push / pull code to / from other repositories.

A number of tools were developed to provide awareness regarding version control systems. Among the ones that notify commit activities, we have *SVN Notifier*[[10]](#footnote-10), *SCM Notifier*[[11]](#footnote-11), *Commit Monitor*[[12]](#footnote-12), *SVN Radar*[[13]](#footnote-13) and *Hg Commit Monitor*[[14]](#footnote-14)*.* All of them focus on avoiding conflicts by increasing the developer’s perception of concurrent work. In addition, they generally are simple tools for which we have not found extensive research or published work, and they fail to identify related repositories and do not provide information in different levels of details, such as status, branches, and commits. DyeVC provides these different levels of details, as shown in Section III.B.

On the other hand, approaches that generally have related research work published are the ones that not only give the developer awareness of concurrent changes, but also inform if any conflicts were detected, such as *Polvo* [22] *Palantir*, [17], *CollabVS* [19], *Crystal* [15], *Safe-Commit* [20], *Lighthouse* [14], and *WeCode* [21]. Among these studies, only *Crystal* works with DVCSs. It detects physical, syntactic, and semantic conflicts (provided that the user informs the compile and test commands), but does not deal with repositories that pull updates from more than one partner and demands having a Git client installed. *Polvo* presents metrics that quantify merging complexity involving Subversion branches, which are calculated reactively (upon user request). Although DyeVC primary focus is not to detect conflicts, it can be seen as a supporting infrastructure that can be combined with such approaches to allow conflicts and metrics analysis over DVCS.

FASTDash [18] does not detect conflicts, as the previous cited studies, but provides awareness of potential conflicts, such as two programmers editing the same region of the same source file. It uses a spatial representation of the shared codebase, highlighting team members’ current activities. Unfortunately, the tool was designed for project teams of 3-8 developers, which makes it inappropriate to be used in large projects.

In terms of the Commit History visualization in DyeVC, which is a kind of repository linearization, existing research have contributed several techniques, with different focuses. Among these techniques, we can cite:

* Gevol [29], which focuses on program structures;
* Evolution Matrix [30], which focuses on classes;
* CVSscan [31], a line oriented tool, where columns represent different versions of a file and the horizontal direction is used for time;
* CodeSaw [32], a tool that compares authors’ code and project mail contributions along one year, exposing their peaks and valleys of productivity;
* VisGi [33], VRCS [34] and Visugit [35], and GitHub’s network graph, which focus on the branch history.

Most of these repository linearization techniques were applied only to CVCSs. The only exception found were VisGi, Visugit and GitHub’s network graph, which work with Git repositories, but look only at a local repository, not showing, for example, where a given commit can be found.

Regarding the topology view in DyeVC, we could not find any similar work.

# Conclusions and Future Work

Collaborative software development is a great challenge. If, on the one hand, parallelism in activities brings scale gains, on the other hand it tends to increase the concurrency, causing rework and productivity loss. The proliferation of branches and the distribution of repositories makes it difficult to realize parallel actions made by different developers. In this paper, we presented DyeVC, a tool that identifies the status of a repository in contrast with its partners, which are dynamically found in an unobtrusive way.

DyeVC allows one to have different views of a repository. The response time and scalability of each varies according to the information granularity, with greater granularity giving lower performance results.

A number of research topics arise from this approach. Different visualizations can be developed to show the commit history, compacting it, for example, by collapsing contiguous nodes that represent commits with the same level of accessibility. The ability to attach new layouts and filters allows the development of new visualizations, in order to present different metrics and views of the repository (e.g. which repositories or which people changed a specific artifact or group of artifacts, which commits introduced a higher amount of changes in the code, among others). DyeVC could also integrate with tools that give awareness of existing and possible conflicts among work being made concurrently.

References

[1] M. J. Rochkind, “The source code control system,” *IEEE Trans. Softw. Eng. TSE*, vol. 1, no. 4, pp. 364–470, Dec. 1975.

[2] J. Estublier, “Software configuration management: a roadmap,” in *Proceedings of the Conference on The Future of Software Engineering*, New York, NY, USA, 2000, pp. 279–289.

[3] W. Tichy, “RCS: A system for version control,” *Softw. - Pract. Exp.*, vol. 15, no. 7, pp. 637–654, 1985.

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

[5] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato, *Version Control with Subversion*. Stanford, CA, USA, 2011.

[6] S. Chacon, *Pro Git*, 1st ed. Berkeley, CA, USA: Apress, 2009.

[7] B. O’Sullivan, *Mercurial: The Definitive Guide*, 1st ed. O’Reilly Media, 2009.

[8] Eclipse Foundation, “The Open Source Developer Report - 2013 Eclipse Community Survey,” San Francisco, CA, USA, Jun. 2013.

[9] C. Walrad and D. Strom, “The importance of branching models in SCM,” *Computer*, vol. 35, no. 9, pp. 31 – 38, Sep. 2002.

[10] B. O’Sullivan, “Making sense of revision-control systems,” *Commun. ACM*, vol. 52, no. 9, pp. 56–62, Sep. 2009.

[11] D.-C. Gumm, “Distribution Dimensions in Software Development Projects: A Taxonomy,” *IEEE Softw.*, vol. 23, no. 5, pp. 45–51, Sep. 2006.

[12] R. D. Battin, R. Crocker, J. Kreidler, and K. Subramanian, “Leveraging resources in global software development,” *IEEE Softw.*, vol. 18, no. 2, pp. 70–77, Mar. 2001.

[13] D. E. Perry, H. P. Siy, and L. G. Votta, “Parallel changes in large scale software development: an observational case study,” in *Proceedings of the 20th International Conference on Software engineering*, Washington, DC, USA, 1998, pp. 251–260.

[14] I. A. da Silva, P. H. Chen, C. Van der Westhuizen, R. M. Ripley, and A. van der Hoek, “Lighthouse: coordination through emerging design,” in *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, New York, NY, USA, 2006, pp. 11–15.

[15] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, “Proactive detection of collaboration conflicts,” in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, New York, NY, USA, 2011, pp. 168–178.

[16] B. Appleton, S. Berczuk, R. Cabrera, and R. Orenstein, “Streamed lines: Branching patterns for parallel software development,” in *Proceedings of the 1998 Pattern Languages of Programs Conference*, Monticello, Illinois, USA, 1998, vol. 98.

[17] A. Sarma and A. van der Hoek, “Palantir: coordinating distributed workspaces,” in *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*, Oxford, United Kingdom, 2002, pp. 1093 – 1097.

[18] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson, “FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2007, pp. 1313–1322.

[19] P. Dewan and R. Hegde, “Semi-synchronous conflict detection and resolution in asynchronous software development,” in *Proceedings of the 10th European Conference on Computer-Supported Cooperative Work*, 2007, pp. 159–178.

[20] J. Wloka, B. Ryder, F. Tip, and X. Ren, “Safe-commit analysis to facilitate team software development,” in *Proceedings of the 31st International Conference on Software Engineering*, Washington, DC, USA, 2009, pp. 507–517.

[21] M. L. Guimarães and A. R. Silva, “Improving early detection of software merge conflicts,” in *Proceedings of the 2012 International Conference on Software Engineering*, Piscataway, NJ, USA, 2012, pp. 342–352.

[22] R. Santos and L. G. P. Murta, “Evaluating the Branch Merging Effort in Version Control Systems,” in *Proceedings of the 26th Brazilian Symposium on Software Engineering (SBES)*, Natal, RN - Brazil, 2012, pp. 151–160.

[23] C. M. Cesario and L. G. P. Murta, “What is going on around my repository?,” presented at the I Brazilian Workshop on Software Visualization, Evolution and Maintenance, Brasília, Brasil, 2013, pp. 14–21.

[24] J. I. Maletic, A. Marcus, and M. L. Collard, “A task oriented view of software visualization,” in *First International Workshop on Visualizing Software for Understanding and Analysis, 2002. Proceedings*, 2002, pp. 32–40.

[25] S. Diehl, *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*, 2007 edition. Berlin ; New York: Springer, 2007.

[26] P. Dourish and V. Bellotti, “Awareness and Coordination in Shared Workspaces,” in *Proceedings of the 1992 ACM Conference on Computer-supported Cooperative Work*, New York, NY, USA, 1992, pp. 107–114.

[27] I. Steinmacher, A. Chaves, and M. Gerosa, “Awareness Support in Distributed Software Development: A Systematic Review and Mapping of the Literature,” *Comput. Support. Coop. Work CSCW*, pp. 1–46, May 2012.

[28] C. Gutwin, K. Schneider, D. Paquette, and R. Penner, “Supporting group awareness in distributed software development,” in *Engineering Human Computer Interaction and Interactive Systems*, vol. 3425, R. Bastide, P. Palanque, and J. Roth, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.

[29] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler, “A System for Graph-based Visualization of the Evolution of Software,” in *Proceedings of the 2003 ACM Symposium on Software Visualization*, New York, NY, USA, 2003, p. 77–ff.

[30] M. Lanza, “The Evolution Matrix: Recovering Software Evolution Using Software Visualization Techniques,” in *Proceedings of the 4th International Workshop on Principles of Software Evolution*, New York, NY, USA, 2001, pp. 37–42.

[31] L. Voinea, A. Telea, and J. J. van Wijk, “CVSscan: Visualization of Code Evolution,” in *Proceedings of the 2005 ACM Symposium on Software Visualization*, New York, NY, USA, 2005, pp. 47–56.

[32] E. Gilbert and K. Karahalios, “LifeSource: Two CVS Visualizations,” in *CHI ’06 Extended Abstracts on Human Factors in Computing Systems*, New York, NY, USA, 2006, pp. 791–796.

[33] S. Elsen, “VisGi: Visualizing Git branches,” in *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, 2013, pp. 1–4.

[34] H. Koike and H.-C. Chu, “VRCS: integrating version control and module management using interactive three-dimensional graphics,” in *1997 IEEE Symposium on Visual Languages, 1997. Proceedings*, 1997, pp. 168–173.

[35] T. Hozumi, “Visugit.” [Online]. Available: https://github.com/hozumi/visugit. [Accessed: 04-May-2014].

1. [↑](#footnote-ref-1)
2. 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. [↑](#footnote-ref-2)
3. https://github.com/ [↑](#footnote-ref-3)
4. https://gitorious.org/ [↑](#footnote-ref-4)
5. http://bitbucket.org/ [↑](#footnote-ref-5)
6. http://docs.oracle.com/javase/6/docs/technotes/guides/javaws/ [↑](#footnote-ref-6)
7. http://www.eclipse.org/jgit/ [↑](#footnote-ref-7)
8. http://jung.sourceforge.net/ [↑](#footnote-ref-8)
9. https://www.mongodb.org/ [↑](#footnote-ref-9)
10. http://svnnotifier.tigris.org/ (2012) [↑](#footnote-ref-10)
11. https://github.com/pocorall/scm-notifier (2012) [↑](#footnote-ref-11)
12. http://tools.tortoisesvn.net/CommitMonitor.html (2013) [↑](#footnote-ref-12)
13. http://code.google.com/p/svnradar/ (2011) [↑](#footnote-ref-13)
14. http://www.fsmpi.uni-bayreuth.de/~dun3/hg-commit-monitor (2009) [↑](#footnote-ref-14)