What is going on around my repository?

Cristiano M. Cesário, Leonardo G. P. Murta

Instituto de Computação
Universidade Federal Fluminense (UFF) – Niterói, RJ – Brazil
{ccesario, leomurta}@ic.uff.br

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

1 Introduction

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 evolved from a centralized repository with local access, as in SCCS and RCS [Tichy 1985], to a client-server approach, as in CVS [Cederqvist 2005] and Subversion [Collins-Sussman et al. 2011]. More recently, distributed VCSs (DVCS) arose, allowing clones of the entire repository in different locations, as in Git [Chacon 2009] and Mercurial [O'Sullivan 2009a]. According to a survey conducted among the Eclipse community [Eclipse Foundation 2012], Git and Github usage increased from 13% to 27% between 2011 and 2012 (a growth greater than 100%). Between 2010 and 2011, this growth had already been around 78% [Eclipse Foundation 2011]. This clearly shows momentum and a strong tendency in the adoption of DVCSs among the open source community.

According with [Walrad and Strom 2002], 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. DVCSs include better support to work 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, 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.

The proliferation of branches, the increasing growth of development teams, and their distribution along distant locations – even different continents –, introduce additional complexity to notice 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 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. 2011] 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. 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 repository. This scenario generates traffic similar to that of peer-to-peer applications. With this diversity of topologies, the administration of 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?" or "What are the dependencies between different clones?", among others.

Most of existing works deal with the last two issues, giving to the developers the perception of concurrent changes. Palantir [Sarma and Van der Hoek 2002], Lighthouse [Da Silva et al. 2006], CollabVS [Dewan and Hegde 2007], Safe-Commit [Wloka et al. 2009], Crystal [Brun et al. 2011], WeCode [Guimarães and Silva 2012], and Polvo [Santos and Murta 2012] 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 introduces DyeVC[1], which is a novel visualization infrastructure for DVCS. The main goal of DyeVC is to increase the developer knowledge of what is going on around his repository and the repositories of his teammates. DyeVC is an extensible tool that 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 rest of this paper is organized as follows. Section 2 presents a motivational example to this work. Section 3 presents the approach used in DyeVC, along with its current state of development. Section 4 discusses some related work and in Section 5 we conclude the paper and present future work.

2 Motivational Example

Figure 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. Arrows in Figure 1 indicate the direction in which updates are sent. Thus, for example, Rogue can pull updates from Gambit, and both Gambit and Beast can pull updates from Rogue.

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

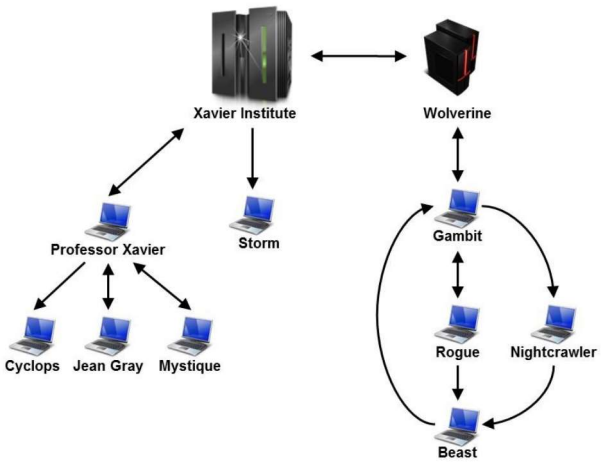

Figure 1 – A development scenario involving some developers

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.

Taking Beast as an example, at a given moment, how can he know 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?

3 DyeVC

The approach we propose with DyeVC involves continuously monitoring a group of interrelated repositories, based on repositories registered by the user. The implementation uses Java Web Start[2] Technology, and focus on monitoring Git repositories. The gathering of information from repositories is accomplished using JGit[3] library, which allows the user to use DyeVC without having a Git client installed. DyeVC presents a visual log using JUNG[4] 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 it presents.

DyeVC gathers information in different levels of detail, presenting it in a visual style, and supplying notifications whenever there are changes in any of the registered repositories or in its partners, which are repositories that a given repository communicates with, as shown in Figure 2. The period between subsequent monitor runs is configurable and defaults to 5 minutes.

---

[2] http://docs.oracle.com/javase/6/docs/technotes/guides/javaws/

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

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

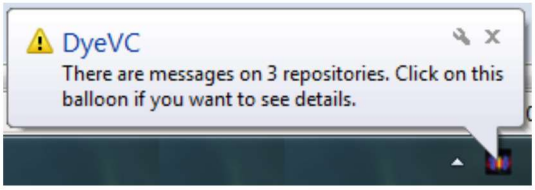

Figure 2 – DyeVC showing notifications in the notification area

The levels of detail defined in DyeVC include: presenting the status of a given repository against its partners (Level 1); zooming into the branches 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).

In Level 1, DyeVC presents the status of a repository against its partners. The status presented by DyeVC can be one of those presented in Table 1.

Table 1 – Possible status of a repository

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

The evaluation of the status individually considers the existing commits in each repository. 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 2.

Table 2 – 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	Ahead and Behind (needs push and *pull*)
Yes	No	Ahead (needs *push*)
No	Yes	Behind (needs *pull*)
No	No	Synchronized

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 Figure 1 have the commits shown in Table 3.

Table 3 – 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 Figure 1, which depend on the direction of the arrows, the perception of each developer regarding to his known partners is shown in Table 4. 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 4 - Status of each repository based on known remote repositories

Repository	Wolverine	Gambit	Rogue	Nightcrawler	Beast
Wolverine	-	✓	-	-	-
Gambit	✓	-	⇅	-	↑
Rogue	-	⇅	-	-	-
Nightcrawler	-	↑	-	-	-
Beast	-	-	↓	↓	-

The main window of DyeVC presents Level 1 information, as shown in Figure 3. 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).

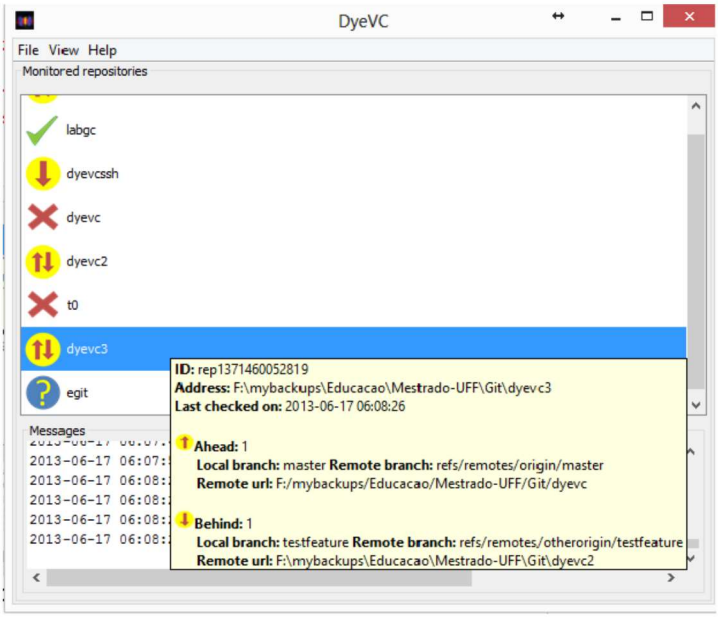

Figure 3 – DyeVC Main Screen

Level 3 information is shown as a visual log of the repository (Figure 4). Each node in the graph represents a commit, and receives a color according with its type, which can be one of: regular node (cyan), branch node (red), merge node (green), branch and merge node (yellow), start node (black) or branch head (last commit in a branch, in dark gray). Nodes 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 log window can also be zoomed in or out, whether the user wants to see details of a particular area of the log 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.

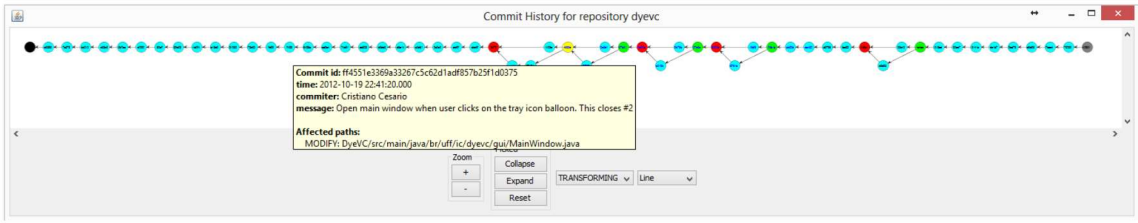

Figure 4 – Log Window (commit history)

4 Related Work

DyeVC relates primarily with studies that aim at improving the perception of developers that work with distributed software development (awareness tools). A recent work by [Steinmacher et al. 2012] presents a systematic review of such studies. We can group these works in two categories: those that notify commit activities (conflict avoidance) and those that detect conflicts (conflict detecting).

Approaches that notify commit activities, such as *SVN Notifier*[5], *SCM Notifier*[6], *Commit Monitor*[7], *SVN Radar*[8] and *Hg Commit Monitor*[9] focus on avoiding conflicts by increasing the developer's perception of concurrent work. However, 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 3.

Approaches that detect conflicts, such as *Polvo* [Santos and Murta 2012] *Palantir*, [Sarma and Van der Hoek 2002], *CollabVS* [Dewan and Hegde 2007], *Crystal* [Brun et al. 2011], *Safe-Commit* [Wloka et al. 2009], *Lighthouse* [Da Silva et al. 2006], and *We-Code* [Guimarães and Silva 2012] not only give the developer awareness of concurrent changes, but also inform if any conflicts were detected. Among these works, 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. On the other hand, *Polvo* presents metrics that quantify merging complexity between involving Subversion branches, which are calculated reactively (upon user request). DyeVC can be seen as a supporting infrastructure that can be combined with such approaches to allow conflicts and metrics analysis over DVCS.

---

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

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

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

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

[9] http://www.fsmpi.uni-bayreuth.de/~dun3/hg-commit-monitor (2009)

5 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.

A number of research topics arise from this approach. The ability to discover partners of a repository induces that it is possible to find all existing clones of a repository, showing them as a network topology of interrelated nodes that communicate with each other, sending and receiving updates. A global commit history view could be drawn, showing all existing commits in all nodes in the network, allowing one to see which commits exist somewhere, and which ones have not been propagated to all nodes yet. 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 high amount of changes in the code, which branches would cause a conflict if merged, among others).

Acknowledgments

The authors would like to thank CNPq and FAPERJ for the financial support.

References

Appleton, B., Berczuk, S., Cabrera, R. and Orenstein, R. (Aug 1998). Streamed lines: Branching patterns for parallel software development. In *Proceedings of the 1998 Pattern Languages of Programs Conference.* , PLoP 1998. ACM.

Brun, Y., Holmes, R., Ernst, M. D. and Notkin, D. (Sep 2011). Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering.* , ESEC/FSE '11. ACM.

Cederqvist, P. (2005). *Version Management with CVS.* Free Software Foundation.

Chacon, S. (2009). *Pro Git.* 1. ed. Berkeley, CA, USA: Apress.

Collins-Sussman, B., Fitzpatrick, B. W. and Pilato, C. M. (2011). *Version Control with Subversion.* Stanford, CA, USA: .

Da Silva, I. A., Chen, P. H., Van der Westhuizen, C., Ripley, R. M. and Van der Hoek, A. (Oct 2006). Lighthouse: coordination through emerging design. In *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange.* , eclipse '06. ACM.

Dewan, P. and Hegde, R. (Sep 2007). Semi-synchronous conflict detection and resolution in asynchronous software development. In *Proceedings of the 10th European Conference on Computer-Supported Cooperative Work.* , ECSCW 2007. Springer London.

Eclipse Foundation (Jun 2011). The Open Source Developer Report - 2011 Eclipse Community Survey.

Eclipse Foundation (Jun 2012). The Open Source Developer Report - 2012 Eclipse Community Survey.

Estublier, J. (May 2000). Software configuration management: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering.* , ICSE '00. ACM.

Guimarães, M. L. and Silva, A. R. (Jun 2012). Improving early detection of software merge conflicts. In *Proceedings of the 2012 International Conference on Software Engineering.* , ICSE 2012. IEEE Press.

O'Sullivan, B. (2009a). *Mercurial: The Definitive Guide*. 1. ed. O'Reilly Media.

O'Sullivan, B. (Sep 2009b). Making sense of revision-control systems. *Communications of the ACM*, v. 52, n. 9, p. 56–62.

Perry, D. E., Siy, H. P. and Votta, L. G. (Apr 1998). Parallel changes in large scale software development: an observational case study. In *Proceedings of the 20th International Conference on Software engineering.* , ICSE '98. IEEE Computer Society.

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

Santos, R. and Murta, L. G. P. (2012). Evaluating the Branch Merging Effort in Version Control Systems. In *Proceedings of the 26th Brazilian Symposium on Software Engineering (SBES).* , SBES '12. IEEE Computer Society.

Sarma, A. and Van der Hoek, A. (Aug 2002). Palantir: coordinating distributed workspaces. In *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International.*

Steinmacher, I., Chaves, A. and Gerosa, M. (May 2012). Awareness Support in Distributed Software Development: A Systematic Review and Mapping of the Literature. *Computer Supported Cooperative Work (CSCW)*, p. 1–46.

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

Walrad, C. and Strom, D. (Sep 2002). The importance of branching models in SCM. *Computer*, v. 35, n. 9, p. 31 – 38.

Wloka, J., Ryder, B., Tip, F. and Ren, X. (May 2009). Safe-commit analysis to facilitate team software development. In *Proceedings of the 31st International Conference on Software Engineering.* , ICSE '09. IEEE Computer Society.