tABLE OF cONTENTS

[Chapter 1 – Introduction 2](#_Toc406785189)

[1.1 Motivation 2](#_Toc406785190)

[1.2 Goals 4](#_Toc406785191)

[1.3 Research Questions 5](#_Toc406785192)

[1.4 Contributions 5](#_Toc406785193)

[1.5 Organization 5](#_Toc406785194)

[Chapter 2 – Conclusion 7](#_Toc406785195)

[2.1 Contributions 7](#_Toc406785196)

[2.2 Limitations 7](#_Toc406785197)

[2.3 Future work 8](#_Toc406785198)

# – Introduction

## Motivation

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, 2009b). According to a survey conducted among the Eclipse community (ECLIPSE FOUNDATION, 2013), 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.

Besides these changes from local to client-server and then to distributed architecture, the concurrency control policy adopted by VCSs also changed from lock-based (pessimistic) to branch-based (optimistic). According with Walrad and Strom (WALRAD; STROM, 2002), creating branches in VCSs is essential to software development because it enables concurrent development, allowing the maintenance of different versions of a system in parallel, the customization to different platforms and to different customers, among other features that are expected by current software development teams. DVCS include better support to work with branches (O’SULLIVAN, 2009a), 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. All these branches, whether explicit or not, will eventually be reintegrated by means of merge operations, reflecting to the main development line the changes made.

However, distributed software development, especially from the geographical perspective (GUMM, 2006), brings a set of risk factors, and Configuration Management is affected by them (BATTIN *et al.*, 2001). The increasing growth of development teams, and their distribution along distant locations – even different continents – together with the proliferation of branches, introduce additional complexity for perceiving actions performed in parallel by different developers. According to Perry *et al.* (PERRY *et al.*, 1998), concurrent development increases the number of defects in software. Besides, Silva *et al.* (DA SILVA *et al.*, 2006) say that branches are frequently used for promoting isolation amongst developers. 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.* (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, DVCS expand the branching possibilities discussed by Appleton *et al.* (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. In practice, projects impose some restrictions over this topology freedom. However, it can be still much more complex than the traditional client-server topology found in Centralized Version Control Systems (CVCS).

To illustrate this situation, Figure 1 shows a scenario with some developers, each one owning a clone of the repository originally created at Xavier Institute. Xavier Institute acts like a central repository, where code developed by all teams is integrated, tested, and released to production. There is a team working at Xavier Institute, led by Professor Xavier, and a remote developer (Storm) that periodically receives updates from the Institute. Outside the Institute, Wolverine leads a remote team located in a different site, which is constantly synchronized with the Institute. Solid lines in Figure 1 indicate data being pushed, whereas dotted lines indicate data being pulled. Thus, for example, Rogue can both pull updates from Gambit and push updates to him, and Beast can only pull updates from Rogue.



Figure 1 - A development scenario involving some developers

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

As an example, from a developer’s point of view, like Beast, how can he know at a given moment if there are commits in Rogue, in Gambit, or in Nightcrawler clones that were not pulled yet? Alternatively, would be the case that there are local commits pending to be pushed to Gambit? Beast could certainly periodically pull changes from his peers, checking if there were updates available, but this would be a manual procedure, prone to be forgotten. It would be more practical if Beast could have an up to date knowledge of his peers, warning him about any local or remote updates that had not been synchronized yet.

On the other hand, from an administrator’s point of view, how can she know the existing clones of a project and how they relate among each other? How can she know if there are pending commits to be sent from a staging repository to a production one? This kind of perception regarding others work is known as “awareness”, which is defined by (DOURISH; BELLOTTI, 1992) as “an understanding of the activities of others to provide a context for one’s own activities”.

Most of the existing approaches that deal with providing awareness of concurrent work (in different clones or different branches) are focused only in CVCSs, which are much less prone to branches if compared to DVCSs. Other approaches focus on DVCSs, but looking at a specific branch and without offering a way to discover dependencies between clones (i.e., peers), or changes introduced in different branches of work.

Therefore, this work main motivation is to establish an extensible platform that helps DVCS administrators and users in understanding who the existing peers are and how they relate with each other.

## Goals

In this work, we propose a novel visualization infrastructure for DVCS, which gathers information about different clones of a repository and presents them visually to the user, allowing one to perceive how his repository evolved over time and how this evolution compares to the evolution of other repositories in the project.

Thus, this work proposes an extensible platform that enables repository administrators to visualize which the existing repositories of a project are and how they interact with each other. Having this information is important to verify if communication is taking place accordingly, based on what was defined in the CM Plan.

This work also aims at increasing the developer knowledge of what is going on around his repository and the repositories of his teammates, despite the branch where changes are being done.

## Research Questions

Our approach and its evaluation have the primarily objective to answer the following questions:

* Q1: Which clones were created from a repository?
* Q2: What are the dependencies between 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 my work?
* Q4: Is it computationally feasible to gather this information from all known repositories, keeping them available to be used when needed?

## Contributions

This work introduces a new infrastructure for DVCS monitoring and awareness that can gather information from DVCS repositories, consolidate this information and provide a series of extensible visualizations to the user. These visualizations can help administrators and developers in knowing who the participating peers in a project are and how they depend upon each other. The infrastructure also opens new research possibilities to enhance the existing visualizations and provide new ones. The information that is gathered can be increased and used to mine information in the repositories and thus uncovering usage patterns or presenting metrics.

## Organization

Besides this introduction, this work is organized in four other chapters. <Chapter 2> presents some introductory topics regarding DVCS. It contrasts DVCS usage against CVCS. It also explains the concept of *branches* and how they are used in DVCS. Lastly, it presents the related work, which include commit visualization approaches, approaches that provide awareness of concurrent changes, and approaches that focus on repository visualization.

<Chapter 3> presents the approach, named DyeVC[[1]](#footnote-1). This chapter describes how DVCS information is gathered and structured. Then it outlines the existing visualizations in a hierarchical way, discussing the level of detail included in each one, over the example introduced in Figure 1. It also discusses the algorithm used in information gathering, which is in the heart of the process that discovers related peers, dependencies, and work in progress. Furthermore, it presents the technologies used in the implementation. Finally, it shows a typical usage scenario of DyeVC, describing its prototype and the first steps needed to use it.

<Chapter 4> describes the evaluation performed on the usage of DyeVC to provide awareness over an open source project that uses DVCS. Next, the scalability of the approach is evaluated, presenting the factors that may affect the capability of using DyeVC. Lastly, it presents some threats to the validity of the performed evaluation.

Finally, <Chapter 5> concludes this work, presenting contributions, limitations, and future work.

# – Conclusion

## Contributions

This work introduced a new approach for DVCS monitoring and awareness, entitled *DyeVC*. This approach gathers information from registered DVCS clones and their peers, regarding the flow of communication and the existing commits in every node, and records this information in a central database.

The gathered information is consolidated, allowing developers to increase their knowledge of what is going on that might affect their work, as well as which changes have to be sent/received to/from their teammates. It also gives repository administrators the knowledge about which are the existing clones of a project and how they interact with each other.

*DyeVC* shows the information in different levels of detail, from a high-level topology-like visualization, where each node represents a repository clone, to a detailed level that presents every commit, despite the repository where it is located. Most of the visualizations use an extensible graph library that allows the approach to be extended through the creation of new visualizations and filters. The visualizations use transformations to present vertices and edges using different icons, colors, line types, and text labels, according to the characteristics that we want to highlight. This way, we established a framework for coupling different visualizations related to DVCS.

We have evaluated DyeVC on a real project, showing that it can be used to answer questions that arise when working with DVCSs. We have also evaluated DyeVC’s performance when used with 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 under analysis, especially in the view levels with finer granularity.

## Limitations

*DyeVC* has a scalability limitation, regarding processing performance and memory usage. We use the *Dijkstra* algorithm provided by the JUNG graph library to minimize the number of crossing lines in the lower level visualization (that shows each commit in the topology), and this procedure is not optimized to deal with graphs that contain thousands of nodes, although the high level topology visualization can be used on repositories with many thousands of commits.

Another limitation is related to the need of a central database to record information gathered from the several *DyeVC* instances. Although this central database is needed, we used a document-based database that is hosted free in the Internet, and the information is read and written using semi-structured JSON documents, that are automatically mapped to/from the application class model.

The need of a central database brings another limitation in the current implementation, regarding to security. The connection with the central database is authenticated by using an application key that is stored in the application itself. This way, different projects from different organizations will have their data gathered and stored in the same database. Although we do not store any sensitive information (we do not store contents of any files or commits, just metadata), this might be a concern for some people. We have plans to create different adapters, for example, to store the gathered information in local databases, with a per user authentication.

## Future work

The advent of *DyeVC* approach brings with it a number of possibilities for future researches. The following paragraphs describe possible improvements and researches that can be explored in the future.

The first improvement is related to the visualizations the approach already provides. Level 4 visualization, which shows every commit in the topology, could be enhanced with automatic collapsing of similar nodes. Currently, each vertex in level 4 visualization represents a single commit. Depending on the repository size, this leads to a graph that is very long horizontally, because we show each commit on a different X-coordinate, to give the idea of elapsed time. Even with the zooming feature, large repositories can be difficult to analyze. It happens that we normally want to analyze the very ending part of a repository, which comprises of the most recent commits in the topology, because the older ones probably were spread to the whole topology already. The current implementation has a feature for the user to select a group of commits and manually collapsing them, creating a single node that represents the group of collapsed commits, which is placed at the midpoint between the first and the last collapsed nodes. However, on a repository with thousands of commits, this is not very practical. Automatic collapsing could compact the visualization, by collapsing contiguous nodes that represent commits with the same level of accessibility, leaving only branch heads expanded.

Another possible improvement related to level 4 information is to attach filters and transformations to help answering a number of user questions, such as: Which repositories or people changed a specific artifact or group of artifacts? Which commits introduced the higher amount of changes in the code? Who were the top contributors in the project this week?

By increasing the amount of metadata that *DyeVC* already gathers, a number of research options arise. For example, supposing that we are dealing with text artifacts, if *DyeVC* gathers the changes introduced by each commit at the line level (by storing each commit’s *diff*), one could create a visualization to show conflicts that would happen when merging two branches.

Another area demanding work is related to scalability. Currently, all vertex and edges must be loaded into memory in order to calculate vertices positions when drawing level 4 information. Besides that, for repositories with a large number of commits, more than a minute is spent to calculate positions and draw the graph. The usage of automatically collapsing could help in terms of time spent to draw the graph, as there would be less vertices and edges to be plot, but the memory issue would be still be present, because the collapsed nodes would be still loaded into memory. A possible way to solve that would be to filter commits before plotting them, for example showing only commits performed this month. A downside of this approach is that it could lead to a disconnected graph, for example, if work has been done over this month on two separate branches whose common ancestor is a commit performed a long time ago, we would see two parallel sequence of commits, with no common ancestor.

A possible improvement in Level 2 visualization (which shows the topology) is regarding how the approach registers existing clones. In this visualization, once registered, clones will be presented forever. It might be the case that one had just registered a clone with *DyeVC* and never worked on it again. After some time, this could lead to a polluted topology view, with lots of “garbage”, i.e. repositories that are not used or that might not even exist. The approach could check when was the last change in each clone, marking those clones that did not change for a period time, so that an administrator could remove it from the topology. Similarly, an administrator could manually include nodes in the topology, to represent clones located in places with no *DyeVC* instance running, in order to complete the topology not previously seen by the approach.

Bibliography

APPLETON, B.; BERCZUK, S.; CABRERA, R.; ORENSTEIN, R. Streamed lines: Branching patterns for parallel software development. In: PATTERN LANGUAGES OF PROGRAMS CONFERENCE (PLOP 98), Aug. 1998, Monticello, Illinois, USA: ACM, Aug. 1998.

BATTIN, R. D.; CROCKER, R.; KREIDLER, J.; SUBRAMANIAN, K. Leveraging resources in global software development. *IEEE Software*, v. 18, n. 2, p. 70–77, Mar. 2001.

BRUN, Y.; HOLMES, R.; ERNST, M. D.; NOTKIN, D. Proactive detection of collaboration conflicts. In: ACM SIGSOFT SYMPOSIUM AND EUROPEAN CONFERENCE ON FOUNDATIONS OF SOFTWARE ENGINEERING (ESEC/FSE’11), Sep. 2011, New York, NY, USA: ACM, Sep. 2011. p. 168–178.

CEDERQVIST, P. *Version Management with CVS*. [N.A.]: Free Software Foundation, 2005.

CHACON, S. *Pro Git*. 1. ed. Berkeley, CA, USA: Apress, 2009.

COLLINS-SUSSMAN, B.; FITZPATRICK, B. W.; PILATO, C. M. *Version Control with Subversion*. Stanford, CA, USA: Compiled from r4849, 2011.

DA SILVA, I. A.; CHEN, P. H.; VAN DER WESTHUIZEN, C.; RIPLEY, R. M.; VAN DER HOEK, A. Lighthouse: coordination through emerging design. In: WORKSHOP ON ECLIPSE TECHNOLOGY EXCHANGE (OOPSLA ’06), Oct. 2006, New York, NY, USA: ACM, Oct. 2006. p. 11–15.

DOURISH, P.; BELLOTTI, V. Awareness and Coordination in Shared Workspaces. In: ACM CONFERENCE ON COMPUTER-SUPPORTED COOPERATIVE WORK (CSCW ’92), 1992, New York, NY, USA: ACM, 1992. p. 107–114.

ECLIPSE FOUNDATION. *The Open Source Developer Report - 2013 Eclipse Community Survey*. Survey. San Francisco, CA, USA: Eclipse Foundation, Jun. 2013.

ESTUBLIER, J. Software configuration management: a roadmap. In: INTERNATION CONFERENCE ON SOFTWARE ENGINEERING (ICSE ’00), May 2000, New York, NY, USA: ACM, May 2000. p. 279–289.

GUMM, D.-C. Distribution Dimensions in Software Development Projects: A Taxonomy. *IEEE Software*, v. 23, n. 5, p. 45–51, Sep. 2006.

O’SULLIVAN, B. Making sense of revision-control systems. *Communications of the ACM*, v. 52, n. 9, p. 56–62, Sep. 2009a.

O’SULLIVAN, B. *Mercurial: The Definitive Guide*. 1. ed. Sebastopol, CA, USA: O’Reilly Media, 2009b.

PERRY, D. E.; SIY, H. P.; VOTTA, L. G. Parallel changes in large scale software development: an observational case study. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE 98’), Apr. 1998, Washington, DC, USA: IEEE Computer Society, Apr. 1998. p. 251–260.

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

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

WALRAD, C.; STROM, D. The importance of branching models in SCM. *IEEE Computer*, v. 35, n. 9, p. 31 – 38, Sep. 2002.

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. [↑](#footnote-ref-1)