Semistructured Merge on Git: An Assessment

Guilherme Cavalcanti
Informatics Center
Federal University of Pernambuco
Recife, Brazil
Email: gjcc@cin.ufpe.br

Paola Accioly
Informatics Center
Federal University of Pernambuco
Recife, Brazil
Email: prga@cin.ufpe.br

Paulo Borba
Informatics Center
Federal University of Pernambuco
Recife, Brazil
Email: phmb@cin.ufpe.br

***Abstract*—The conflict resolution strategies adopted by Version Control Systems (VCSs) can be initially classified as unstructured and structured. Recently, researchers proposed a third approach, the semistructured approach, that tries to combine the strengths of the other ones. As there is little evidence of the benefits and drawbacks of each approach, in this paper we present a study that empirically compares the unstructured and semistructured approaches over the Git, a natively unstructured VCS. We reproduced 154 merge scenarios from 10 open source projects developed in C#, Java and Python and we found that the semistructured approach can substantially reduce the number of conflicts, in particular, it was able to resolve ordering conflicts - conflicts that rely on software's elements order (methods, imports, fields, and so on).**

I. Introduction

In the context of software design, modularity is the system's property of being structured in modules or components of code that have well-defined scope and responsibilities [1]. In theory, the more modular a system, the easier it will be its development and maintenance because modules can be viewed as independent units and developed in parallel. However, it is difficult to model a structure of modules that provides and successfully encapsulates the different types of changes that may occur during the life cycle of a software. For example, in software product lines and systems using Feature-Driven Development or other agile methods, it is common requests for changes involving the addition of new functionalities, also called *features*, which can interact with each other intentionally or accidentally. What happens in practice is that when two or more teams implement features in parallel they need to modify pieces of code that intersect and end up causing integration conflicts.

During parallel software development, the occurrence of conflict emerges as a major problem because developers have inconsistent copies of the shared assets. Such conflicts are costly because they delay the project while the developers investigate and resolve conflicts [2].

A good parallel software development is only possible thanks to a configuration management and the consequent use of VCSs. The conflict resolution strategies adopted by VCSs can be classified into (1) *unstructured*, (2) *structured*, and, recently, a new approach emerged (3) *semistructured* that attempts to combine the strengths of both, the generality of *unstructured* merge and expressiveness of the *structured* merge; the idea of this approach is to provide structural information about software artifacts, so that conflicts are resolved automatically, and when this information is not enough, it applies the usual textual resolution to the conflict [3].

This work aims to conduct an empirical study on the activities of integration and conflict resolution in software projects that use the Git as VCS. Nowadays, Git is one of the most popular and widespread VCS and is natively *unstructured*. We want to compare the *unstructured* approach with the *semistructured* one. To this end, both approaches must be used in a number of merge scenarios and, finally, we compare the resulting number of conflicts.

II. Background and related work

The purpose of a VCS is to manage different revisions of a software system. With the use of VCSs, developers create a revision of the software system from a base version and can develop and evolve it separately, and finally integrate it back into the base version when conflicts typically occurs. Zimmermann conducted an empirical study and showed that 23% to 46% of merge scenarios presents conflicts [4]. Conflicts can appear in different forms, among them, conflicts that arise due to parallel changes in the same artifact, the so-called *direct* conflict [2], [5]. Regarding the strategies to resolve direct conflicts, VCSs can be unstructured or structured.

Unstructured VCSs operates purely in plain text or tokens. If two revisions modify or extend the text in the same region, the system notifies conflict. That is, it is not able to decide how to merge the modifications or extensions.

In turn, structured VCSs are specific to a programming language and uses information inherent to the artifacts of the language to resolve the largest possible number of conflicts automatically [6].

By analyzing *unstructured* and *structured* approaches, we observe that there is a relationship between generality and expressiveness, unstructured VCSs are very general, they can be used with any type of textual content, but are not able to resolve conflicts that require specific knowledge of the language of that content. On the other hand, structured VCSs are very expressive, they are typically specific to a particular language. Based on these observations, Apel et al. questioned if it was possible to develop a VCS able to combine these features and proposed the *semistructured* approach.

The *semistructured* approach represents software artifacts as trees and provide information (through an annotated grammar)

about how nodes of certain types (methods, classes, etc.) and its subtrees can be merged (via superimposition [7]).

The ability of *semistructured* merge to resolve certain conflicts is based on the observation that the order of certain elements (classes, interfaces, methods, fields, imports, and so on) does not matter - the so-called *ordering conflicts* [3]. By abstracting the document structure as tree, *semistructured* merge has enough information to identify ordered items. When the *semistructured* merge doesn't know how to merge a software element, like method bodies with statements, it represents the elements as plain text and uses the conventional *unstructured* merge. It also allows special conflict handlers to be added, to resolve specific cases of conflicts. Thus, the approach becomes a combination of the *unstructured* and *structured* approaches. The *semistructured* merge is more expressive than the first one, because it solves conflicts automatically based on the information available about the language; and more general than the second, since it supports a larger number of languages by simply providing an annotated grammar of the language to be supported.

To evaluate the *semistructured* approach, Apel et al. conducted a similar empirical study on 23 projects using the Subversion as VCS, analyzing a total of 180 merge scenarios, and found that the *semistructured* approach was able to reduce (on average) the number of syntactic conflicts, conflicting lines of code, and conflicting files in 34%, 61% and 28% respectively.

III. Evaluation Study

Our objective is to analyze the merging process of different software revisions to compare two conflict resolution approaches (unstructured and semistructured) with respect to their ability to resolve conflicts automatically and assess how many conflicts would be resolved automatically if Git was a semistructured VCS.

We use a sample of 154 merge scenarios from 10 software projects that uses the Git as VCS obtained from GitHub, written in Java, Python and C#. The projects were chosen based on (1) trending[1] of GitHub, (2) number of contributors and (3) number of commits.

In this experiment, we apply both approaches to the same merge scenarios and we analyze the number of syntactic conflicts, conflicting lines of code and conflicting files resulting from each approach.

The merging process of the revisions involves the integration of two branches/revisions together with a common ancestor revision, in a process called *three-way merge* [8]. Thus, a merge scenario is a set consisting of the base, left and right revisions, where the base revision is the moment from which the left and right revisions have been derived.

IV. Results and contributions

In our merge scenarios, on average, the *semistructured* approach was able to reduce in 86.76% the number of syntactic conflicts, 92.28% the amount of conflicting lines of code and 85.4% the amount of conflicting files in relation to the *unstructured* approach. We can also say that many of the conflicts that occur in merging revisions are ordering conflicts, which can be resolved automatically with semistructured merge. All material about the experiment is available online[2].

We also found isolated cases where *unstructured* approach presented fewer conflicts, which in principle is counter-intuitive given the nature of the *semistructured* merge that behaves like the unstructured merge where it is not able resolve conflicts automatically. Apel at al. demonstrated that this occurs in cases of renaming, which poses a challenge to the *semistructured* approach. This happens because it uses superimposition to merge revisions. If a component of a program is renamed in a revision (in one of the branches), the merge algorithm is not aware of this fact and cannot map the renamed component in its earlier version, this results in a situation where a branch has an empty or missing component.

These results showed that the semistructured approach can substantially reduce the occurrence of conflicts, either in number of syntactic conflicts, conflicting lines of code or conflicting files. Even if there are situations in which the semistructured approach increases the number of syntactic conflicts and conflicting lines of code, especially in the presence of renaming, where the semistructured approach produces more conflicts, but smaller (fewer lines of code) than in unstructured.

Acknowledgment

We would like to thank CNPq and CAPES/PROCAD Brazilian research funding agencies and INES, funded by CNPq and FACEPE, for partially supporting this work.

References

[1] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[2] 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*. ACM, 2011, pp. 168–178.

[3] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, "Semistructured merge: rethinking merge in revision control systems," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 190–200.

[4] T. Zimmermann, "Mining workspace updates in cvs," in *Mining Software Repositories, 2007. ICSE Workshops MSR'07. Fourth International Workshop on*. IEEE, 2007, pp. 11–11.

[5] B. K. Kasi and A. Sarma, "Cassandra: Proactive conflict minimization through optimized task scheduling," in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 732–741.

[6] T. Mens, "A state-of-the-art survey on software merging," *Software Engineering, IEEE Transactions on*, vol. 28, no. 5, pp. 449–462, 2002.

[7] S. Apel and C. Lengauer, "Superimposition: A language-independent approach to software composition," in *Software Composition*. Springer, 2008, pp. 20–35.

[8] B. O'Sullivan, "Making sense of revision-control systems," *Communications of the ACM*, vol. 52, no. 9, pp. 56–62, 2009.

[1] https://github.com/trending

[2] http://goo.gl/eBR4eS