Measuring the effectiveness of structure-aware diffing

Experience report

Giovanni Garufi
University of Utrecht
Utrecht, The Netherlands
nazrhom@gmail.com

Victor Cacciari Miraldo
University of Utrecht
Utrecht, The Netherlands
v.cacciarimiraldo@uu.nl

Wouter Swierstra
University of Utrecht
Utrecht, The Netherlands
w.s.swierstra@uu.nl

ABSTRACT

Today's version control systems rely on the Unix *diff* utilities to detect which lines in a file have been changed and to merge different changes to the same file. Not all such changes, however, are best represented in terms of modifications to lines of code. This may lead to unnecessary conflicts that must be resolved manually by developers. This paper explores the usage of an alternative algorithm for merging the *syntax trees* of the programming language Clojure. As a result, a significant number of conflicts drawn from existing Clojure repositories may be merged automatically, providing evidence that tree-based algorithms offer better precision than the traditional line-based approach in determining which changes give rise to conflicts.

CCS CONCEPTS

•**Software and its engineering** → *Software configuration management and version control systems; Software version control; Collaboration in software development;*

KEYWORDS

Conflict resolution; merge conflicts; diff algorithms; Clojure

1 INTRODUCTION

Version control systems are indispensable in modern software development. Most implementations of version control, such as mercurial [9, 12] or git [14] , are built using the Unix *diff* utility. This utility compares two files line-by-line and computes a *patch*, describing how to transform one file into another. When a given file has been modified in two different ways (possibly by two different developers), the corresponding changes must be reconciled somehow. *Merging* patches may fail: if the same line has been modified in different ways, it is not clear which one of the two modifications should be preferred. As a result, developers must resolve the resulting *conflict* manually.

While *diff* is tried and tested, there are certain scenarios where merging patches produced using the *diff* utility fails—even though a successful merge strategy may exist. To illustrate this issue, consider the following function written in a Lisp-like language:

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
PPDP'18, Frankfurt, Germany
© 2018 Copyright held by the owner/author(s). 978-x-xxxx-xxxx-x/YY/MM...$15.00
DOI: 0000001.0000001

```
(defn head [l]
  (first l))
```

Suppose there are two independent changes made to this function: the first adds a default parameter to be returned in case the list is empty; the second renames the function from head to fst. Consider these changes side by side:

```
(defn head [l, d]            (defn fst [l]
  (if (nil? l)                 (first l))
      (d)
      (first l)))
```

When attempting to reconcile these two changes line-by-line we will run into problems. Although the two patches modify separate parts of code, merging will result in a conflict: two different changes occur on the same line.

Tools such as *diff* rely on a fixed granularity of change: the lines of code that are modified. While this works well for many languages, examples such as the one presented here have motivated the exploration of alternative algorithms comparing the underlying *abstract syntax trees* of the code under version control. In fact, for each conflict deduced from a line-based patch, one of three situations can occur when considering the underlying *abstract syntax tree*: there is no conflict (as the two changes modife different parts of the tree); the conflict could be easily solved automatically using a heuristic; or there is a real conflict (and the same part of the abstract syntax tree is changed in two different ways). The fixed granularity of *diff* has other drawbacks. For example, some version control systems refuse to merge changes on adjacent lines. Merging such adjacent changes may introduce unexpected interaction between the different modifications; requiring at least one unchanged line between two changes provides a limited guarantee that the changes effect independent parts of the code.

The head function given above is a good example of a situation where there really is no conflict. Joining the two changes with a *syntax-aware* tool would result in the following code:

```
(defn fst [l , d]
  (if (nil? l)
      (d)
      (first l)))
```

An automatically solvable conflict, on the other hand, is when the *same* piece of the *abstract syntax tree* has been changed in two different ways. However, knowledge about *which* part of the *AST* has changed enables one to resolve the conflict. We illustrate such a conflict in Figure 1, where the original file, Figure 1c, was changed in two different ways, Figures 1a and 1b. Nevertheless, in the only place where the changes overlap, both change the identifier `bump-version` to `bump-version-map`. The two patches coincide on the overlapping changes. Hence, we can automatically produce the merged file, Figure 1d. This example, along with Figure 2, are taken from Clojure repositories on GitHub.

There are many other situations where conflicts may be resolved automatically using semantic information. For instance, when parsing configuration files, we could solve conflicts on version number of dependencies by always choosing the larger version number. In order to conceive such strategies, it is paramount to have access to the type structure in the AST. One must know the conflict happens on a version number of a dependency to apply this strategy.

Real conflicts, on the other hand, occur when there is an inherent need for human intervention to converge on a single version of the *AST*. Take the example in Figure 2, where a whole portion of a function was changed in two different ways. In general, these types of conflicts will always require human interaction to be solved.

It is worth mentioning that, although it is possible to have conflicts in our *abstract syntax tree* setting that are *not* conflicts in a line based setting, these are extremely rare. They only occur when the same *AST* element span across multiple lines and was changed in two different ways. One example would be multi-line strings. For all practical purposes, we will consider that these conflicts do not exist.

This experience report aims to quantify and document the effectiveness of a tree-structured merge algorithm. We would like to know the frequency and distribution of these different kinds of conflicts. By mining popular Clojure repositories, identifying merge points and conflicts and attempting to merge with an *AST* merging algorithm, we can attempt quantify how many real conflicts occur within Clojure projects. We conjecture that Clojure is a suitable language for study as, in contrast to many imperative languages, its syntax is not line-based. Our algorithms and tooling has all been implemented in Haskell, and is available online. [1]

2 METHODOLOGY

We have collected data from twenty popular Clojure repositories. For each repository, we counted the number of *merge points*. A merge point indicates that files have been changed in two different ways, requiring a merge to reconcile the changes. For each merge point, we distinguish between those merge points that are are solved automatically by *diff*3, and those that give rise to conflicts that must be resolved manually. For each conflict, we check whether the line changes effect disjoint parts of the Clojure syntax trees involved. More specifically, we collected our data as follows:

(1) For each of the repositories we listed in Table 1 we listed all the merge point revisions:
    `git rev-list --merges HEAD`.
(2) For each merge point, we identified the two parent commits involved:
    `git log -1 --format=%P mergepoint`.
(3) Next, we checked out the the first parent, and attempt a merge with the other
    `git checkout -q fstparent && git merge --no-commit other`.
(4) For each resulting merge, we checked whether or not it lead to a conflict:
    `git ls-files --unmerged`.
(5) For each relevant git object, we extract three files: the original file (`O.clj`), and the two modified files (`A.clj` and `B.clj`):
    `git cat-file -p gitobj`.

[1] https://github.com/nazrhom/vcs-clojure

Running the associated script yields a series of directories, each containing three Clojure files. To speed up our experiment, we removed top-level expressions not associated with the conflict from each file. At this point, we could parse the Clojure files, compare the resulting syntax trees, and identify any conflicts at this level.

Tree-structured diff. We have implemented a simple parser for Clojure in Haskell [13]. After parsing each file, we compute a tree-structured diff between the original (`O.clj`) and two derived files (`A.clj` and `B.clj`). To compute this diff, we have implemented a tree-structured diff algorithm for our Clojure abstract syntax tree [11]. The original algorithm described in Agda is *data type generic*: in this case study, we have instantiated the algorithm to work on our Clojure abstract syntax tree and implemented it in Haskell [5].

Transcribing the algorithm to Haskell used a variety of non-trivial Haskell extensions. For one, we have to mimic Agda's dependent types using custom *singleton types*. Furthermore, we rely on many of Haskell's extensions for type-level programming to mimic the computations done in our Agda types. For each constructor of our syntax tree, we introduce a separate singleton type to associate constructors with their type explicitly. The algorithm itself uses a couple of heuristics to speed up the search space. For instance, we use the result of *diff*3 to prune certain choices when deciding whether a part of the abstract syntax tree should be deleted, inserted or copied.

Running our algorithm results in a richly structured patch, describing how to map one Clojure syntax tree into another. For each conflict point that we have identified, we try to merge the two patches involved with the structural merging algorithm and record the results.

We classified each result in one of three ways: (A) structural merging gives no conflicts, as a line was changed in two different ways, effecting disjoint parts of the syntax tree. As a result, *diff*3 signaled a *false conflict*; (B) the structure-aware merge algorithm managed to resolve the conflict automatically, we call these *resolved* conflicts; or (C) even merging syntax trees would require human intervention to resolve the conflict. This last category are what we classify as *true conflicts*.

3 RESULTS

Table 1 shows the results of our experiment. The first columns describe the number of contributors (Contributors), lines of Clojure code (LOC), the total number of commits (Commits), the number of conflicts that *diff*3 has encountered (Conflicts). Note that for some repositories, only counting the Clojure code present leads to skewed statistics. For example, the **circleci**/frontend repository

```
(deftest version-map->string-valid
  (doseq [[string parsed bumps] version-values]
    (doseq [[level string] bumps]
      (is (= (merge :qualifier nil
                    (bump-version-map parsed level))
             (parse-semantic-version string))))))
```

(a) Change A (commit d8765c)

```
(deftest version-map->string-valid
  (doseq [[string parsed bumps] version-values]
    (doseq [[level string] bumps]
      (is (= (merge :qualifier nil
                    (bump-version-map level parsed))
             (parse-semantic-version string))))))
```

(b) Change B (commit d96da3)

```
(deftest version-map->string-valid
  (doseq [[string parsed bumps] version-values]
    (doseq [[level string] bumps]
      (is (= (merge :qualifier nil
                    (bump-version level parsed))
             (parse-semantic-version string))))))
```

(c) Original File, `leiningen` repository

```
(deftest version-map->string-valid
  (doseq [[string parsed bumps] version-values]
    (doseq [[level string] bumps]
      (is (= (merge :qualifier nil
                    (bump-version-map parsed level))
             (parse-semantic-version string))))))
```

(d) Merged File

Figure 1: Resolved Conflict Example

```
(defn prep [project]
  ;; This must exist before the project is launched.
  ...
  (doseq [task (:prep-tasks project)]
    (main/apply-task (main/lookup-alias task project)
                     (dissoc project :prep-tasks) []))
  (.mkdirs (io/file (:compile-path project "/tmp"))))
```

(a) Change A (commit `066b55`)

```
(defn prep [project]
  ;; This must exist before the project is launched.
  ...
  (prep-tasks project)
  (.mkdirs (io/file (:compile-path project "/tmp"))))
```

(b) Change B (commit `07cd35`)

```
(defn prep [project]
  ;; This must exist before the project is launched.
  ...
  (doseq [task (:prep-tasks project)]
    (main/apply-task task (dissoc project :prep-tasks) []))
  (.mkdirs (io/file (:compile-path project "/tmp"))))
```

(c) Original File, `leiningen` repository

Figure 2: Real Conflict Example

contains only a fraction of Clojure code, compared to the other languages used.

The last three columns, however, are the most interesting. These record the cases where *diff*3 reported a conflict unnecessarily, i.e., when a line is modified in two different ways, but these changes effect different parts of the syntax tree (A). Secondly, we record the conflicts that could be resolved automatically, when merging the changes at the level of syntax trees (B). Finally, there are situations where a *true conflict* occurs, where even the structure-aware merge fails (C).

The results show 170 *true conflicts*, these conflicts occur when, for example, a string constant is changed in two different ways. Without further knowledge of the context, it is impossible to merge such changes automatically. The automatic, syntax-directed merge algorithm we employed is parametrised by a function that merges the data stored in leaves of the syntax trees. We can provide this merging strategy with further domain specific knowledge about how to reconcile such changes. For example, when the conflicting strings are version numbers, one viable strategy may be to pick the most recent version. We found 117 conflicts which could potentially have been resolved by automatic strategies. We aim to explore such strategies in future work. Finally we found 165 *false conflicts*, cases where a structural-merge is possible but *diff*3 signals a conflict. We see that out of the conflicts we merged, we could automatically solve 62% more conflicts than *diff*3.

4 CONCLUSIONS

From these numbers, we can see that the despite the large number of commits, conflicts are still fairly rare. Of the tens of thousands of commits we considered, less than 500 resulted in a conflict. Despite these numbers, it is clear that structure-aware diff and merge algorithms manipulating syntax trees gave rise to significantly fewer conflicts than line-based diff algorithms. Of the 452 conflicts we inspected, only 170 could not be resolved using alternative algorithms. This provides evidence that employing and developing structure-aware algorithms is a worthwhile pursuit.

There are a few caveats associated with these numbers. Firstly, the structure-aware diff and merge algorithms are significantly slower than their *diff* and *diff*3 counterparts. The finer granularity of change that the structure-aware algorithms may observe results in a significantly larger search space of patches between trees. In 172 cases, the algorithms failed to find or merge patches within

Name	Contributors	LOC	Commits	Conflicts	A	B	C
marick/Midje	35	14,693	2,416	18	8	2	8
ztellman/aleph	62	4,557	1,064	17	6	5	6
boot-clj/boot	66	9,370	1,271	8	2	2	4
nathanmarz/cascalog	43	8,028	1,366	46	17	14	15
dakrone/clj-http	109	5,193	1,111	5	1	0	4
metosin/compojure-api	36	6,604	1,818	12	1	4	7
wit-ai/duckling-old	65	28,790	586	12	3	2	7
cemerick/friend	33	803	227	1	1	0	0
circleci/frontend	92	894	18,857	27	5	2	20
incanter/incanter	82	16,478	1,282	40	9	22	9
jonase/kibit	47	1,099	401	4	2	0	2
bhauman/lein-figwheel	86	6,515	1,464	6	4	0	2
technomacy/leiningen	315	10,669	4,484	28	12	4	12
clojure-liberator/liberator	42	2,965	347	8	6	1	1
onyx-platform/onyx	46	23,778	6,641	90	46	11	33
overtone/overtone	55	27,935	2,996	50	21	6	23
pedestal/pedestal	59	1,1206	1,403	24	13	5	6
quil/quil	34	1,341	960	10	1	8	3
riemann/riemann	114	16,586	1,654	6	3	0	3
ring-clojure/ring	99	4,909	958	40	4	31	5
Total				452	165	117	170

Table 1: The results collected

the one minute time-out we provided. In future work, we aim to optimize the naive algorithm implemented further in order to investigate these cases.

A second caveat we should mention is that we *only* consider the results of a `git merge` operation. Our mining methodology cannot observe conflicts that occur during a `git rebase`. As `git rebase` rewrites history, the conflicts that may have been there originally are lost to our analysis. This could explain the relatively low number of conflicts found considering the total number of of commits we analyzed.

We have restricted our study to a single programming language, Clojure. It is still unclear if similar studies targeting other languages would produce similar results. We believe that structure-aware diff algorithms have the biggest potential in functional languages, such as Clojure, Haskell, or OCaml [7], where code consists of expressions that may be split into lines in very different ways. In imperative languages, on the other hand, the most common unit of code is a statement; programmers typically have a single statement per line of code. To repeat this study for other languages would require some work, such as writing a parser together with diff and merge algorithms on the syntax trees – yet doing so would provide further insight into *how* code in different languages is organized.

We have not done a quantitative study into the *quality* of conflicts produced. When resolving conflicts on long lines of code, it can be difficult to find the difference between two patches and merge these manually. Structure-aware algorithms, on the other hand, may be useful when resolving conflicts: better precision might help humans *identify* changes and choose between them. In particular, we would like to investigate how to map tree-structured patches to common programming refactorings, such as method renaming, method moving, or method extraction. We believe that being able to present conflicts and changes at a higher level of abstraction than the lines of code that have been changed, will provide valuable insight into understanding how code evolves, and how conflicts arise and may be resolved.

Although several different tree-based diffing and merging algorithms exist [1–4, 6, 8, 10, 15, 16], the aim of this experiment was never to compare *different* algorithms, but rather compare one such algorithm to the current approach underlying today's version control systems. Moreover, only one of the aforementioned algorithms [6, 15] is both *typed* and *generic*. That is, the algorithm can be applied to different programming languages and guarantees that applying a patch produces well-formed syntax trees. Repeating the same experiment with other diff and merge algorithms may indeed produce different results. Crucially, however, the results we have collected demonstrate that the current generation of tools still leave ample room for improvement.

REFERENCES

[1] Apel, S., Liebig, J., Brandl, B., Lengauer, C., and Kästner, C. 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* (New York, NY, USA, 2011), ESEC/FSE '11, ACM, pp. 190–200.

[2] Bergroth, L., Hakonen, H., and Raita, T. A survey of longest common subsequence algorithms. In *Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)* (Washington, DC, USA, 2000), SPIRE '00, IEEE Computer Society.

[3] Bille, P. A survey on tree edit distance and related problems. *Theor. Comput. Sci. 337*, 1-3 (June 2005), 217–239.

[4] Chawathe, S. S., and Garcia-Molina, H. Meaningful change detection in structured data. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1997), SIGMOD '97, ACM.

[5] Garufi, G. Version control systems: Diffing with structure. Master's thesis,

University of Utrecht, the Netherlands, 2018.

[6] Lempsink, E., Leather, S., and Löh, A. Type-safe diff for families of datatypes. In *Proceedings of the 2009 ACM SIGPLAN workshop on Generic programming* (2009), ACM, pp. 61–72.

[7] Leroy, X., Doligez, D., Garrigue, J., Rémy, D., and Vouillon, J. The Objective Caml system release 3.11. *Documentation and user's manual. INRIA* (2008).

[8] Lessenich, O., Apel, S., and Lengauer, C. Balancing precision and performance in structured merge. *Automated Software Engineering 22*, 3 (Sep 2015), 367–397.

[9] Mackall, M. Towards a better SCM: Revlog and mercurial. *Proc. Ottawa Linux Sympo 2* (2006), 83–90.

[10] Mens, T. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng. 28*, 5 (May 2002), 449–462.

[11] Miraldo, V. C., Dagand, P.-É., and Swierstra, W. Type-directed diffing of structured data. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development* (2017), ACM, pp. 2–15.

[12] O'Sullivan, B. *Mercurial: The Definitive Guide.* O'Reilly Media, Inc., 2009.

[13] Peyton Jones, S. *Haskell 98 language and libraries: the revised report.* Cambridge University Press, 2003.

[14] Torvalds, L., and Hamano, J. Git: Fast version control system. *http://git-scm.com* (2010).

[15] Vassena, M. Generic diff3 for algebraic datatypes. In *Proceedings of the 1st International Workshop on Type-Driven Development* (2016), ACM, pp. 62–71.

[16] Zhang, K., and Shasha, D. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput. 18*, 6 (Dec. 1989), 1245–1262.