2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)

Enhancing Precision of Structured Merge by Proper Tree Matching

Fengmin Zhu, Fei He and Qianshan Yu
School of Software, Tsinghua University
Key Laboratory for Information System Security, MoE
Beijing National Research Center for Information Science and Technology Beijing, China
Email: zfm17@mails.tsinghua.edu.cn, hefei@tsinghua.edu.cn, yqs17@mails.tsinghua.edu.cn

***Abstract*—Recently, structured merge has shown its advantage in improving the merge precision over conventional line-based, unstructured merge. A typical structured merge algorithm consists of matching and amalgamation on abstract syntax trees. Existing tree matching techniques aim to figure out optimal matches by maximizing the number of matched nodes. From real-world codebases, however, we find that many of the reported conflicts are unnecessary. We propose a new objective function for defining a *proper* tree matching, with which the overall conflicting rate can be greatly reduced. We conducted experiments on 3,687 merge scenarios extracted from 18 open-source projects. Results show significant merge precision enhancement of our approach.**

***Index Terms*—Revision control systems, software merge, structured merge, tree matching**

I. Introduction

Software merging is a central task in contemporary software development. The conventional unstructured merge, due to its efficient line-based comparing algorithm and the strength of language-independence, has been cooperated in version control systems like `CVS`, `subversion`, and `git`. However, since these unstructured approaches regard programs as lines of plain text merely and neglect the syntactic information, they are unable to resolve merge conflicts precisely. By contrast, structured approaches exploit the language-specific knowledge and hence resolve more conflicts [1].

Structured merge relies on the context-free syntax of the programs and represents them as abstract syntax trees. A typical structured merge algorithm consists of matching and amalgamation on trees. First, a tree matching algorithm outputs a set of matchings between two trees' nodes. Then, a tree amalgamation algorithm performs the actual merge on each pair of the matched nodes and generates the merge result. In the tree matching pass, the goal is to find the "best" matching set between two trees' nodes, which could be regarded as an optimization problem. Existing techniques [2], [3] identify the matching set that maximizes the overall number of matched nodes, or equivalently, minimizes the presence of missed matches. However, the *quality* of the matches are ignored, i.e. the matched nodes may be logically unrelated to each other. Consequently, the yielded matching set is very likely to cause *unnecessary* conflicts, i.e. conflicts that can be avoided if we use a "better" matching strategy.

This research is supported in part by the National Natural Science Foundation of China under Grant No. 61672310 and Grant No. 61527812.

In this paper, we propose a new objective function for tree matching. This function takes the entire tree structure below two nodes into account when evaluating the match between them. We then define a *proper* tree matching to be a matching set that maximizes our objective function. In the tree amalgamation pass, we guarantee the actual merge is performed only between properly matched nodes. In this way, the overall conflicting rate can be reduced.

II. Proper Tree Matching

In structured merge, programs are represented by abstract syntax trees. We assume each node of the tree is labeled with a grammar type, e.g. if-statement, method call, etc. Two nodes u and v are said *similar*, written $u \sim v$, if their labels match. Tree matching is performed between two trees Γ_1 and Γ_2, to find a *matching set* χ between them. A matching set is a set of pairs of nodes, one from each tree, such that:

1) matched nodes must be similar,
2) a node can match at most one node in the opposing tree, and
3) the parent-child relationship as well as the order between sibling nodes are respected.

The objective of the classical tree matching algorithm is to maximize the number of matched nodes. The *quality* of the matches is, however, not taken into consideration. We now present our new objective function $Q_\chi(u, v)$. Let $\chi_{uv} \subseteq \chi$ be a submatching of χ between the children nodes of u and v. Assume u has m children nodes and v has n. $Q_\chi(u, v)$ is recursively defined as follows:

1) if $u \not\sim v$, then $Q_\chi(u, v) = 0$;
2) if $u \sim v$ and both are leaf nodes, then $Q_\chi(u, v) = 1$;
3) otherwise, $Q_\chi(u, v) = \alpha + \frac{\sum_{(a,b)\in\chi_{uv}} Q_\chi(a,b)}{\max\{m,n\}} \cdot (1 - \alpha)$

where α ($0 \leq \alpha \leq 1$) is a *normalization factor*.

In the third case of the above definition, $Q_\chi(u, v)$ is composed of two parts: the part from the root nodes, and another part from the children nodes. These two parts are normalized using a factor α. We fix it as 0.5, giving both parts equal weights. For any pair of nodes $(a, b) \in \chi_{uv}$, $Q_\chi(a, b)$ is recursively calculated. The equation $\frac{\sum_{(a,b)\in\chi_{uv}} Q_\chi(a,b)}{\max\{m,n\}}$ estimates the quality of the matching between the children nodes of u and v. Especially, if one of the two nodes (u and v) is a leaf, χ_{uv} is then empty, thus $Q_\chi(u, v) = \alpha$.

2574-1934/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE-Companion.2019.00117

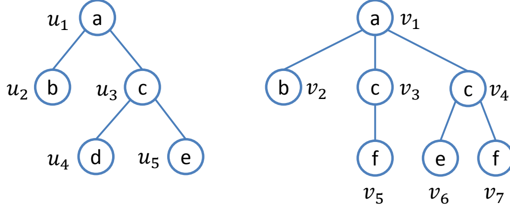

Fig. 1. Two trees

This function is level-aware. Let a, b be the children nodes of u, v. As shown in the above equation, $Q_\chi(a, b)$ contributes to the calculation of $Q_\chi(u, v)$, after being normalized with $1 - \alpha$. Let c, d be the grandchildren nodes of u, v, $Q_\chi(c, d)$ also contributes to the calculation of $Q_\chi(u, v)$, after being normalized twice. In fact, all offspring nodes of u, v have impacts on $Q_\chi(u, v)$, but the deeper level the offspring node is at, the less impact it has on the calculation.

Definition 1. *Let* Γ_1, Γ_2 *be two trees,* the proper tree matching problem *is to find a matching set* χ *to maximize* $Q_\chi(\Gamma_1, \Gamma_2)$.

For example, consider the two trees depicted in Fig. 1. Apparently, u_1 is matched to v_1, u_2 is matched to v_2, and u_3 can be matched to v_3 and v_4. If u_3 is matched to v_3 (such a matching set is named χ_1), none of their children are matched, thus $Q_{\chi_1}(u_3, v_3) = \alpha = 0.5$. Otherwise, if u_3 is matched to v_4 (such a matching set is named χ_2), among their children, only u_5 can be matched to v_6 with the quality $Q_{\chi_2}(u_5, v_6) = 1$. Thus $Q_{\chi_2}(u_3, v_4) = \alpha + \frac{1}{2} \cdot (1 - \alpha) = 0.75$. Finally, our tree matching algorithm returns the matching set $\chi_2 = \{(u_1, v_1), (u_2, v_2), (u_3, v_4), (u_5, v_6)\}$ with the quality $Q_{\chi_2}(u_1, v_1) = \alpha + \frac{1+0.75}{3} \cdot (1 - \alpha) \approx 0.79$.

Let χ^* be the matching set returned by our tree matching algorithm. In company with each node pair in χ^*, its quality is also recorded. One can set a *quality threshold* θ, and use θ to filter the node pairs in χ^*. A node pair $(u, v) \in \chi^*$ is said "properly matched" if $Q_{\chi^*}(u, v) \geq \theta$. Only properly matched nodes are proceeded during the tree amalgamation pass.

III. Implementation and Evaluation

We implemented the proposed proper tree matching algorithm as a tool called AutoMerge, built on top of JDime.

So far, we have conducted experiments on 18 open-source Java projects across different application domains, collected from Github. We analyzed the commit histories and identified the merge commits performed by the developers. From each of them, we extracted a *three-way merge scenario*. Overall, we extracted 3,687 merge scenarios, with totally 131,811 files on which the merge needs to be performed.

We applied JDime and AutoMerge on each of the merge scenarios. We set $\theta = 0.5$. In total, JDime reported 1,915 conflicts, whereas AutoMerge only reported 714, that is, 62.72% of the conflicts were reduced. On 15 of 18 projects, at least one conflict was eliminated. In 2 projects, all of the conflicts reported by JDime were avoided.

Then, we compared the merge results with the ones provided by the developers. We recognized that 98.86% of the results yielded by AutoMerge exactly reproduced the developers' merge, higher than that of JDime (93.12%). AutoMerge reproduced more merges than JDime for all projects. In particular, it reproduced all on 4 projects. Generally speaking, as the quality threshold θ augments, fewer conflicts are reported. However, if θ becomes too high (0.9), the yielded merge is very likely to be rejected by the developer.

Finally, although our new objective function makes the optimization problem harder, the execution time was still acceptable – 3043.66 ms per file, 16.6% slower than JDime.

IV. Related Work

Westfechtel [4] and Buffenbarger [5] pioneered in proposing merge algorithms collaborating the context-free structures of programs. Based on JDime [2], Lessenich et al. [6] also propose a looking ahead mechanism for tree matching, allowing nodes at different levels to be matched. This mechanism identifies more missed matches, however, improper matches are still unnoticed. Our approach is applicable for filtering them out. Furthermore, our approach could also enhance the precision of semi-structured merge [7] and operational-based merge [3], by employing our objective function as a quality measure for the yielded matches.

V. Conclusion

We realize that the existing tree matching algorithm of structured merge leads to many unnecessary conflicts. To eliminate them, we propose a new objective function which takes the quality of the matches into consideration. The goal of our proper tree matching algorithm is to maximize the new objective function. On an evaluation of 18 open-source software projects, we discovered that conflicts were significantly reduced and the overall merge precision was enhanced.

References

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

[2] O. Leßenich, S. Apel, and C. Lengauer, "Balancing precision and performance in structured merge," *Automated Software Engineering*, vol. 22, no. 3, pp. 367–397, Sep 2015.

[3] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 313–324.

[4] B. Westfechtel, "Structure-oriented merging of revisions of software documents," in *Proceedings of the 3rd International Workshop on Software Configuration Management*, ser. SCM '91. New York, NY, USA: ACM, 1991, pp. 68–79.

[5] J. Buffenbarger, "Syntactic software merging," in *Software Configuration Management*, J. Estublier, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 153–172.

[6] O. Leßenich, S. Apel, C. Kästner, G. Seibt, and J. Siegmund, "Renaming and shifted code in structured merging: Looking ahead for precision and performance," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct 2017, pp. 543–553.

[7] 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*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 190–200.