Multidiszciplináris tudományok, 10. kötet. (2020) 3 sz. pp. 365-376 **https://doi.org/10.35925/j.multi.2020.3.44**

MERGING PROBLEMS IN MODERN VERSION CONTROL SYSTEMS

Nasraldeen Alnor Adam Khleel
PhD student, Institute of Information Science, University of Miskolc
3515 Miskolc, Miskolc-Egyetemváros, e-mail: nasr.alnor@uni-miskolc.hu

Károly Nehéz
associate professor, Institute of Information Science, University of Miskolc
3515 Miskolc, Miskolc-Egyetemváros, e-mail: aitnehez@uni-miskolc.hu

Abstract
During software development, when developers change the same part of the code concurrently, this may be led to merging conflicts. Resolving these conflicts might be costly and time-consuming. Three types of conflicts may arise during merge processes: textual, syntactic, and semantic. Textual conflicts occur when merging a concurrent operation, such as addition, removal or edition take place over the same parts of code. Syntactic conflicts occur when concurrent operations break the syntactic structure of the source code files when merged. Finally, a semantic conflict occurs when the merged modification is compiled without error but malfunctions. Version management systems usually use textual merging technique; users can synchronize their modifications with other users working in parallel with them, in this process, a merge is performed between local modifications and remote modifications. The previous work has examined different mechanisms to detect and resolve conflicts and proposed different tools for resolving merge conflicts, such as two-way merging, three-way merging, state-based merging, and operation-based merging. This paper discusses and investigates many concepts related to merging conflicts by asking and answering these questions; what are the factors that most affect in a merge conflict, how to avoiding and reducing merge conflicts, how to detecting merge conflicts, and how to resolve them.

Keywords: *version control systems, merge conflicts, git, merge tools, unstructured merge tools, semi-structured merge tools*

1. Introduction

In software projects, where there are several developers collaboratively working on the same project in the same time [3], each developer has a special workspace and shares contributions through a central/local repository, isolating modifications from the others. This enables developers to work more efficiently by promoting parallel development, but when developers decide integrating code, conflicts may be emerging. Merge conflicts often occur when developers modify the same code artifacts and resolving them might be costly and time-consuming, to minimize these problems, it is important to understand how conflict occurrence is affected by many factors [1]. The most version control systems tools deal with textual conflicts, in this case, a conflict arises when more than one developer makes inconsistent modifications in the same source code, where the modifications cannot be merged into the repository until the conflict is resolved [20]. Merge conflicts are frequent, continual, and arise not only as overlapping textual conflicts but also appear as failures in build and test [29]. Distributed Version

Control Systems (DVCSs), such as Git tool and hosting platforms, such as GitHub, facilitates the software development process. Despite these advanced tools, still there merging and integration problems [4]. To better detect and resolve code integration conflicts, researchers have proposed methods that use different strategies to decrease effort and improve the correctness of the integration [2].

2. Merge conflicts characteristics

Literature on merge conflicts classified several techniques to avoid, reduce and detect conflicts. These techniques helps to avoiding merge conflicts by increasing the developer's awareness of the modifications others made to the source code such as FastDash which sends notifications about potential conflicts when developers are changing the same file. Syde consider the source code modifications at Abstract Syntax Tree (AST) level operations to detect conflicts by comparing tree operations, Palantír detects the modifications made by other developers and presentation them in a graphical, non-intrusive manner. WeCode which continuously merges uncommitted and committed changes to detect merge conflicts. Crystal to detect both direct and indirect conflicts. A software development model to reduce conflicts by notifying developers who are working on the same file. During the development of software projects, when developers change the same part of the source code concurrently, this may be led to conflicts [5]. Merge conflicts are common occurrence in huge and distributed software projects, detection and resolve conflicts does not an easy task. In modern version control systems, merge must do by the last developer implemented the commit [20]. In line-based merging approaches, conflicts can be automatically resolved or manually resolved [16]. In Git system, creating parallel branches or cloning an entire project can be done, this feature allows to create special development lines that make developers work isolated [11]. Git uses a textual merging technique; thus users can synchronize their modifications with other users working in parallel with them. In this process, a merge is performed between local modifications and remote modifications. If developers make a lot of modifications in the same part of code, when they incorporate these modifications, Git cannot decide which modify to choose, in this case, the developers need to resolve the conflict manually, which is an error-prone and time-consuming task [4]. Conflicts might be detected in different stages, during merging or testing, since detecting and resolving conflicts often is a difficult task [7]. The researchers presented three types of conflicts that may arise during a merge [23]. Textual, syntactic, and semantic. Textual conflicts occur when merge concurrent operations, such as addition, removal, or edition take place over the same parts of code. Syntactic conflicts occur when concurrent operations break the syntactic structure of the source code, syntactic structure of the source code means the schema or grammar, for example if there is a variable rename by developer and some added lines using that variable by other developer. The merge will probably have an unresolved symbol. Alternatively, this might introduce a semantic conflict by variable hiding. The semantic conflicts occur when concurrent operations break the semantics of the source code when merged, The semantics of the source code can be expressed by the programming language semantics or expected program behavior, for ex-ample, function rename is a relatively obvious case of a semantic conflict. Currently, most version control systems deal with source code files as text. Therefore, merging is done at a textual level [22]. The conflict resolution strategies adopted by version control systems can be classified into unstructured, structured, and semi-structured [10].

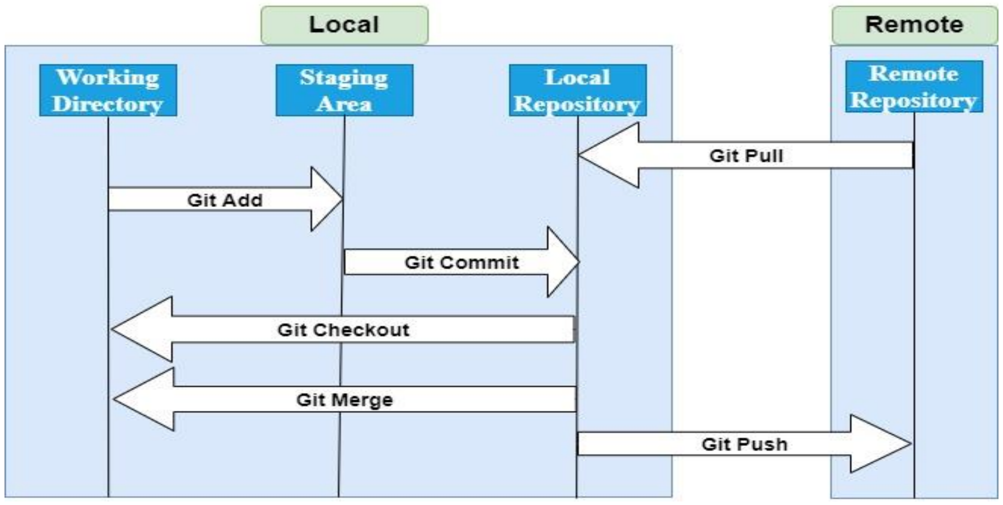


Figure 1. *Common Git Workflow [15]*

3. Merge strategies and tools

Tasks of the software development team are merging code contributions, and therefore, when merging these contributions, conflicts that may occur should be dealt with [2]. Despite the many different merge tools in use today, but they are not perfect, because these tools cannot calculate the simultaneous modifications made by the developers, also some conflicts maybe cannot automatically be resolved, leading to merging fail. So, the developers must intervention to resolve conflicts manually [27]. Therefore, developers manually rely on existing tools to solve merging conflicts [8]. The current merge conflicts tools still rely on structured and unstructured merge strategies. Where the recent developments demonstrate the advanced merge strategies, such as semi-structured merge [2]. The merge techniques based on lines of code as the basis are called unstructured merge techniques. The merge techniques based on syntax and semantics are called structured merge techniques. While the merge techniques that include both aspects of unstructured and structured techniques are called semi-structured techniques [27].

3.1. Structured merge tools

Although version control systems (VCSs) have evolved over the years, merge tools still have some limitations [7]. The structured merge tools are based on syntactic structure and static semantics when merging source code [2]. The goal of structured merge tools is reducing the problems of unstructured merge tools by conflict detection and resolution exploiting the artifacts' structure [26], where uses information inherent to the artifacts of the programming language to solve the conflicts automatically [13]. Structured merging strategy use ASTs level to resolve conflict, an abstract syntax tree (AST) is a way of representing the syntax of a programming language as a hierarchical tree-like structure. This structure is used for generating symbol tables for compilers and later code generation. The tree represents all of the constructs in the language and their subsequent rules [28]. For instance, a syntactic construct like an if-condition-then expression may be denoted by means of a single node with three branches. Where the idea underlying is to represent the artifacts as trees or graphs and to merge them by the tree matching. A structured merge is not only superior in that certain conflict can be resolved automatically, but there are also situations, in which unstructured merge misses' conflicts that are detected by structured merge [26]. The structured merge tools are aware of what kind of source code that deals it, and, therefore, can present better solutions when it comes to conflict handling. But the limitations of structured merge tools are expensive when computing ASTs [28].

3.2. Unstructured merge tools

In unstructured merging tools, merging is done at the text level [12]. Most unstructured merge tools rely on the diff3 algorithm [7]. Many software projects use an unstructured merge strategy because it is very simple and relying on purely textual analysis to detect and resolve conflicts [2]. Where source code written in any programming language is regarded as plain text [8] and every software artifact that can be represented as text. So, each tool can handle the software artifacts conflicts successfully [16]. Ease of use is one of the reasons for the success of unstructured merging tools, applicable to all kinds of text files, and cheap to compute. However, unstructured merge tools have several challenges, because it ignores the syntactic structure of the artifacts during merging [28]. Examples of some famous Unstructured merge tools: *vim merge tool, Meld, Beyond Compare, Araxis Merge, KDiff3, DeltaWalker, P4Merge, Code Compare, and TortoiseMerge* [9].

3.3. Semi-structured merge tools

Semi-structured merge tools are mix unstructured and structured merge tools, it is based on the syntactic structure and static semantics of the artifacts [2]. Semi-structured merge tools represent the elements of source code as trees and use algorithms that to merge nodes and their subtrees [25]. That will be merged based on information about the nodes (methods, classes, etc.). Trees are merged recursively, through superimposition which matches nodes based on structural and nominal similarities [7]. Semi-structured merge tools, for example, *JDime* tool, enhance performance by use ASTs level. A structured merge is then used for the main nodes of the tree, while the unstructured merge is used for the method bodies in the leaves [4]. Semi-structured merge tools have many features, where the merge reports fewer conflicts [2].

4. Compression of merge strategies

To compare merge tools, we should take some aspects such as the measure how often each merge tool can detect interference between development contributions so that it reports interfering changes as conflicts and automatically integrates non-interfering ones, further, when a tool is automatically merges conflicts incorrectly, and when a tool cannot merge trivial changes [25]. The most used software merging tools are unstructured merge tools, it is fast but imprecise as well it more general than structured. In general, the conflicts will not frequent when using both strategies [2]. The semi-structured merge more expressive, because it combine between structured and unstructured, and it resolves conflicts automatically based on the information available about the language, and it supports a larger number of languages by providing an annotated grammar of the language to be supported [13]. The semi-structured merge would be superior because it often reports fewer conflicts, as shown in previous studies [7]. Previous studies compared the merge ways based on concerning the number of reported conflicts, the studies showed semi-structured and structured merge tools have fewer conflicts reports. However, this evidence needs to be further verified. Moreover, a semi-structured and structured merge could even be introducing false positives that might be harder to resolve [27].

Table 1. *Summary of merge conflicts strategies*

Merge strategies	**Examples of tools in strategies**	**How strategies can represent source code**	**How strategies can work**
structured merge	*FSTMerge*	source code is represented as static semantics	analysing the corresponding Abstract Syntax Tree (AST)
unstructured merge	*Meld, Beyond Compare, Araxis Merge, KDiff3, DeltaWalker*	source code is represented as text	each software artifact represented as text
semi-structured merge	*JDime*	source code is represented as static semantics and text	tools represent part of the program elements as trees and use algorithms that know how to merge nodes and their subtrees, this will be done by exploits part of the language syntax and semantics, and based on information about how nodes of certain types (e.g.methods, classes) should be merged. Trees are merged recursively, through superimposition, which matches nodes based on structural and nominal similarities [7]

5. Literature review

In this section, we discuss the previous studies to merge conflicts that most related to our work. In Dias et al. (2020), the authors investigated seven factors related to modularity, size, and timing of developers' contributions, by reproducing and analyse 73504 merge scenarios in GitHub repositories of Ruby and Python MVC projects. The study found that the likelihood of merge conflict occurrence significantly increases when contributions to be merged are not modular. The study also found the bigger contributions involving more developers, commits, and changed files are more likely associated with merge conflicts [1]. In Paikari et al. (2019), the authors suggested *Sayme* as chatbot to detection and resolution of potential source code conflicts that may arise in parallel software development. *Sayme* is designed to informing developers when they do conflicting modifications, and reactively, responding to user inquiries regarding the state of different developers' work and how it may overlap. The study implemented a prototype version of *Sayme* that offers a base layer of functionality, namely the detection of potential direct conflicts as well as a limited form of indirect conflicts [5]. In Fengmin, Zhu et al. (2018) authors proposed an interactive approach for resolving merge conflicts, to represent a very large set of programs proposed an expressive and efficient representation by version space algebra, the study was conducted and evaluated based on 244 real-world conflicts arising from 10 open-source projects. The study found that the *AutoMerge* detects 244 conflicts spread over 138 files, and successfully resolves as high as 95.1% of the conflicts [8]. McKee et al. (2017) authors proposed a study on the factors that impact how practitioners approach merge conflicts and the difficulties they face when resolving conflicts. The study was conducted based on semi-structured interviews on 10 software practitioners across 7 organizations. The study found that the most factors that make difficult merge conflicts are the complexity of conflicting lines of code, the knowledge/expertise in the area of conflicting code, the complexity of the files with conflicts, and the number of conflicting lines of code [21]. In Ahmed et al. (2017), the authors supposed that entities in code smells are lead merge conflict. To get metrics about code modifications and conflicts, analysed 143 repositories from GitHub. The study found that entities that are smelly increase the number of conflicts about three times [23]. In this paper, from literature review about merge conflicts. We investigated how to avoid, reducing, and resolve merge conflicts with git, by answering the following research questions:

RQ1: What are the factors that most influence in merge conflict?
RQ2: How to avoid, reducing merge conflicts?
RQ3: How to detect merge conflicts?
RQ3: How to resolve merge conflicts?

6. Factors that influence merge conflicts

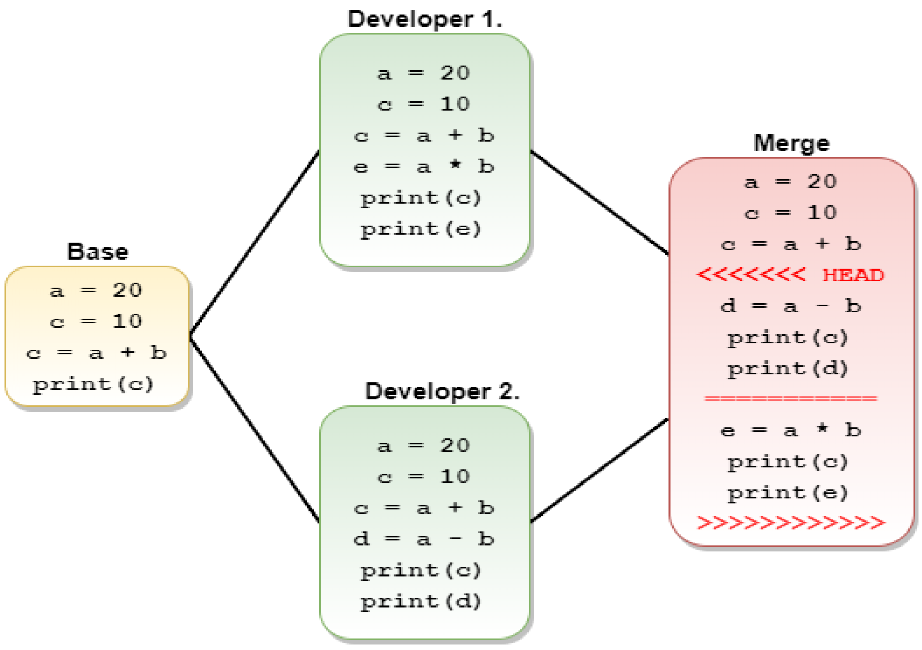
The process to creating a new branch is easy and fast in modern version control systems, but merging is hard and time-consuming, especially when dealing with many branches, so must know the factors that most affect in merge conflict to help developers to avoid and resolve conflicts [18]. When resolving conflicts, the main challenge for developers is understanding the changes that led to the conflict [6]. The largest contribution that includes more developers' pledges and changing files are more likely to relate to consolidation conflicts [1]. Given the cost of merge conflicts and integration problems, many research efforts have advocated earlier resolution of conflicts. Previous work has shown that lack of awareness of modifications being done by other developers can cause conflicts and since infrequent merging can decrease awareness, it increases the chance of conflicts. These conflicts might be detected during merging, building, and testing [14]. These conflicts occur due to many causes. For example, when different developers make modifications to the same artifact without being aware of the other modifications or when there are concurrent modifications in different artifacts, leading to failures in build or test [13].

Table 2. *Factors that most influence in merge conflict [1]*

Number of merge conflicts	Sum of all conflicting chunks in the files in the scenario that reported by the git line-based merge tool when merging the associated contributions, by analyze the files modified by each contribution in a merge scenario. So, modification many files will lead to more conflicts.
Number of files with merge conflicts	The number of files with at least one merges conflict reported by the git line-based merge tool. So, the modifications in many files will more effect on merging conflicts than modifications in a few files.
Number of developers	Number of commit authors in each contribution.
Number of commits	Number of commits in each contribution.
Number of changed files	Number of changed files in each contribution.
Number of changed lines	Number of added and removed lines in each contribution.
Duration	Computing the number of days between the last commit in the contribution and the common ancestor with the other contribution for each contribution to be merged.
Conclusion delay	Compute the difference, in days, between the dates of the last commit of each contribution, because delaying the merge increases conflicts.

7. Avoiding, reducing merge conflicts

Today's software projects will be developed and maintain by many developers and source code can be updated from multiple sources, which may lead to a conflict. The manual solution to this conflict is time-consuming and error prone. There are various approaches to resolve this problem, some of them define guidelines to avoid conflict, and others use tools to automatically resolve the conflicts [19]. Although conflicts are not something to be feared (for example in Git), resolving conflicts can require careful analysis and discussion with other team members, which in the end can be quite time-consuming, So, the most strategies dealing with conflicts is to try to avoid conflicts, this means the developers must concentrate on much more frequent and smaller commits to avoid merging conflicts [15]. Conflicts are costly as they delay the development process, in the period between conflicts that occur and they are discovered and understood, they might grow and become difficult to resolve [16]. Most previous works aim at reducing or avoiding merge conflicts and their negative impacts [6]. Software development often requires social interaction. So, previous studies have provided mechanisms to avoid and minimize merge conflicts, for instance, the communication among developers is efficient for avoiding merge conflicts [17].



```
a = 20
c = 10
c = a + b
print(c)
```

```
a = 20
c = 10
c = a + b
e = a * b
print(c)
print(e)
```

```
a = 20
c = 10
c = a + b
d = a - b
print(c)
print(d)
```

```
a = 20
c = 10
c = a + b
<<<<<<< HEAD
d = a - b
print(c)
print(d)
===========
e = a * b
print(c)
print(e)
>>>>>>>>>>>>
```

Figure 2. *Example of a merge conflict between two developers*

8. Git commands to avoiding and reducing merge conflicts

Fetch: fetch command used to download and copies all branch files to your device; it is possible to fetch multiple branches simultaneously. $ Git fetch: git fetch allows you to do a git diff between your local branch and the remote branch spotting potential conflicts in the process, fetching from the repository brings all new branches and tags for remote tracking without incorporating these changes into your branches. *Git config rerere*. enable true: git rerere is reuse recorded resolution when the merge conflict occurs, it records this solve of merged conflict that is used to resolve a new conflict, this means that we don't spend time-solving recurring merge conflicts, once it's enabled if there are conflicts previously registered in the same source code, rerere automatically resolves these conflicts. $ Git pull --rebase origin master: to see the files have a conflict, after that, you can easily solve merge conflict by use merge conflict tools. *Stash*: Stashing has a working directory for track modified files and saves it as unfinished modifications to apply them at any time. $ Git stash: Temporarily stores all modified tracked files, to save the changes before adding to repository. $ Git stash pop: Restores the most recently stashed files [9].

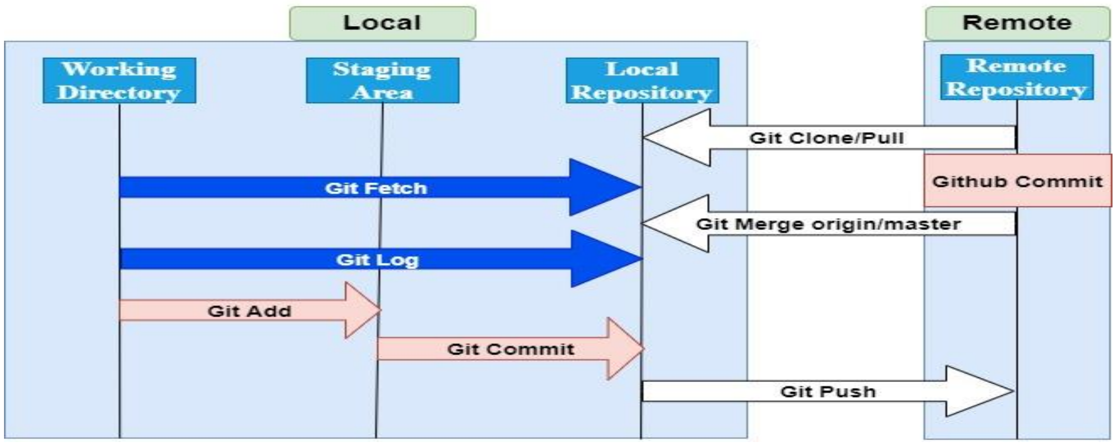


Figure 3. *git fetch scenario*

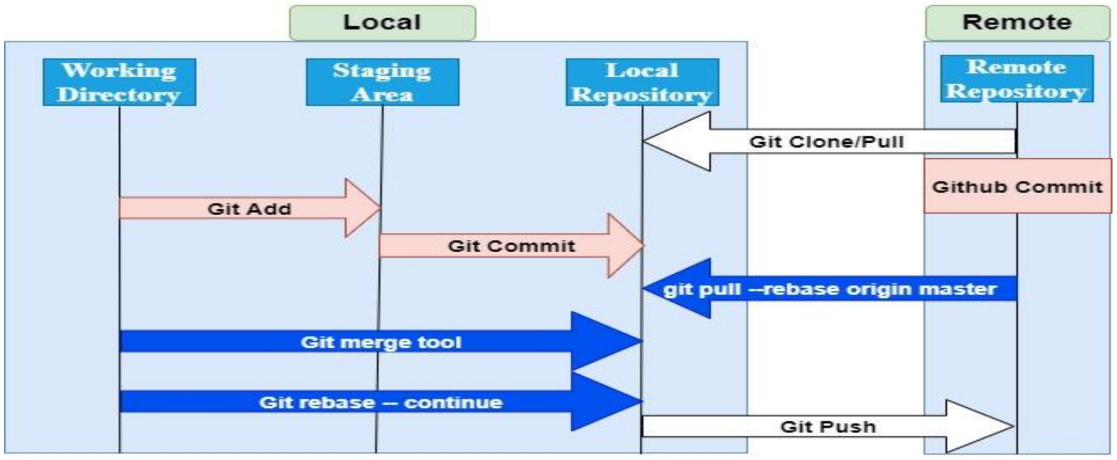


Figure 4. *git pull –rebase scenario*

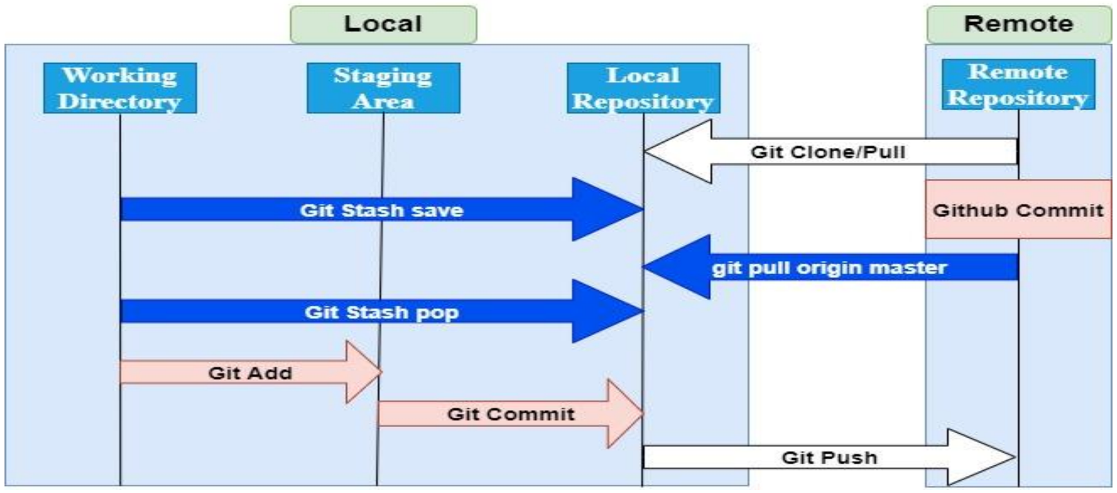


Figure 5. *git stash scenario*

9. Tools to detect merge conflicts

There are different tools to detect conflicts such as two-way merging, three-way merging, state-based merging, operation-based merging, etc. [21, 25], these tools can help to decrease integration effort and improve integration correctness. [14]. To detect conflicts early, there are many tools helps to increase the developer's awareness. Awareness about the modifications team members may be making is a key factor in team productivity and reduces the number of conflicts [4]. These tools monitor all the modifications developers make in their local repositories and after detecting conflicts among those modifications, it will send notifications to the involved developers [5], examples of some tools, *Elvin* and *Syde* are tools supporting direct conflict by increasing awareness through sharing the code change present in other developers' workspaces. *Palantir* and *CollabVS* are tools supporting indirect conflicts by visually illustrates code changes and helps developers avoid conflicts by making them aware of modifications in special workspaces. *Crystal* is a visual tool that uses speculative analysis to help developers detect, manage, and prevent various types of conflicts and taking a different tact in distinguishing textual, build, and test conflicts. *Cassandra* is another tool to minimize conflicts by optimizing task scheduling, to minimize simultaneous edits to the same files. *MergeHelper* is a tool helps developers detect the causes of merge conflicts by providing them with information about historic edit operations. *FastDash* is providing a holistic view of the entire shared workspace [4, 5].

10. How to resolve a merge conflict

Despite there are increasing in merge conflict resolution tools, developers need to resolve conflicts manually and they require expertise to conflict resolution, they also need to understand the changes that led to the conflict and that were done to both the conflicting versions [6]. The main challenge in software integration is conflict resolution, most current integration tools rely on the developer to manually resolve conflicts [8]. Conflict resolution requires a deeper understanding of the source code structure. Prior works have found that in complex merges, developers may not have the expertise or knowledge to make the right decisions, which might degrade the source code quality [11, 23]. If conflicting modifications refer to different lines of the source code, the conflict is automatically resolved by the system, on the contrary, if conflicting modifications refer to the same lines of the source code, the conflict cannot be automatically resolved and the user has to manually resolve it [16]. Version control systems classified the conflict resolution strategies into three which are unstructured, structured, and semi-structured. The structured merge tools are based on programming languages to revolve the merge conflicts. By contrast, unstructured merge tools are based on textual similarity to revolve the merge conflicts. The semi-structured approaches are based on programming languages and textual similarity to revolve the merge conflicts automatically and could extensively reduce the number of conflicts [24].

11. Conclusion

Distributed Version Control Systems, such as Git tool and hosting platforms, such as GitHub, made software development easy, but still there can be merging and integration problems. Version control systems classified the conflict resolution strategies into three approaches, which are: unstructured, structured, and semi-structured. To better detect and resolve code integration conflicts, researchers have proposed tools that use different strategies to decrease effort and improve the correctness of the integration. Despite the many different merge tools in use today, but they are not perfect, because they cannot account for every concurrent modify by developers, also some conflicts maybe cannot automatically be resolved, leading to merging fail. In this paper, many concepts related to merging conflicts has been discussed and we tried to answer some important issues about avoid, reducing, and resolve merge conflicts.

References

[1] Klissiomara, D., Borba, P., Barreto, M.: *Understanding predictive factors for merge conflicts*. Information and Software Technology: 106256, 2020. **https://doi.org/10.1016/j.infsof.2020.106256**

[2] Guilherme, C., et al.: *The impact of structure on software merging: Semi structured versus structured merge*. 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2019. **https://doi.org/10.1109/ASE.2019.00097**

[3] Stanislav, L., Yehudai, A.: *Alleviating merge conflicts with fine-grained visual awareness*. arXiv preprint arXiv:1508.01872, 2015.

[4] Moein, O-K., Nadi, S., Rubin, J.: *Predicting merge conflicts in collaborative software development*. 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, 2019. **https://doi.org/10.1109/ESEM.2019.8870173**

[5] Elahe, P., et al.: *A chatbot for conflict detection and resolution.* 2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE). IEEE, 2019. **https://doi.org/10.1109/BotSE.2019.00016**

[6] Wardah, M. et al.: *Causes of merge conflicts: a case study of Elastic Search*. Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems. 2020. **https://doi.org/10.1145/3377024.3377047**

[7] Guilherme, C., Borba, P., Accioly, P.: *Evaluating and improving semistructured merge*. Proceedings of the ACM on Programming Languages 1.OOPSLA pp.1-27., 2017. **https://doi.org/10.1145/3133883**

[8] Fengmin, Z., He, F.: *Conflict resolution for structured merge via version space algebra*. Proceedings of the ACM on Programming Languages 2. OOPSLA (2018): 1-25. **https://doi.org/10.1145/3276536**

[9] https://hackr.io/blog/git-cheat-sheet., accessed in 2020. April.

[10] Guilherme, C., Accioly, P., Borba, P.: *Semistructured merge on Git: An Assessment*.

[11] Xiaoqian, X., Maruyama, K.: *Automatic software merging using automated program repair*. 2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF). IEEE, 2019. **https://doi.org/10.1109/IBF.2019.8665493**

[12] Caius, B.: *How do developers resolve merge conflicts? An investigation into the processes, tools, and improvements*. Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2018. **https://doi.org/10.1145/3236024.3275430**

[13] Guilherme, C., Accioly, P., Borba, P.: *Assessing semistructured merge in version control systems: A replicated experiment*. 2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, 2015. **https://doi.org/10.1109/ESEM.2015.7321191**

[14] Guilherme, C., Borba, P., Accioly, P.: *Should we replace our merge tools?* 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). IEEE, 2017. **https://doi.org/10.1109/ICSE-C.2017.103**

[15] Tepavac, I., et al.: *Sustavi za verzioniranje, alati i dobra praksa: slučaj Git.*

[16] Hoai L. N., Ignat, C-L.: *An Analysis of Merge Conflicts and Resolutions in Git-Based Open Source Projects*. Computer Supported Cooperative Work (CSCW) 27.3-6 (2018): 741-765. **https://doi.org/10.1007/s10606-018-9323-3**

[17] Gustavo, V., et al.: *On the relation between Github communication activity and merge conflicts*. Empirical Software Engineering 25.1 (2020): 402-433. **https://doi.org/10.1007/s10664-019-09774-x**

[18] Leßenich, O., et al.: *Indicators for merge conflicts in the wild: survey and empirical study*. Automated Software Engineering 25.2 (2018): 279-313. **https://doi.org/10.1007/s10515-017-0227-0**

[19] Diptikalyan, S., et al.: *Delta refactoring for merge conflict avoidance*. Proceedings of the 9th India Software Engineering Conference. 2016. **https://doi.org/10.1145/2856636.2856640**

[20] Costa, C., Figueirêdo, J. JC., Murta, L.: *Collaborative merge in distributed software development: Who should participate?* SEKE. 2014.

[21] McKee, S., et al.: *Software practitioner perspectives on merge conflicts and resolutions*. 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2017. **https://doi.org/10.1109/ICSME.2017.53**

[22] Nelson, N., et al.: *The life-cycle of merge conflicts: processes, barriers, and strategies*. Empirical Software Engineering 24.5 (2019): 2863-2906. **https://doi.org/10.1007/s10664-018-9674-x**

[23] Iftekhar, A., et al.: *An empirical examination of the relationship between code smells and merge conflicts*. 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, 2017. **https://doi.org/10.1109/ESEM.2017.12**

[24] Nazatul N. Z., Ngah, A., Deraman, A.: *Version control system: A review*. Procedia Computer Science 135 (2018): 408-415. **https://doi.org/10.1016/j.procs.2018.08.191**

[25] Tavares, A. T., et al.: *Semistructured merge in JavaScript systems*. 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2019.

[26] Sven, A., Leßenich, O., Lengauer, C.: *Structured merge with auto-tuning: balancing precision and performance*. Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. 2012. **https://doi.org/10.1145/2351676.2351694**

[27] Guilherme, C.: *What merge tool should I use?* Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity. 2017. **https://doi.org/10.1145/3135932.3135943**

[28] Leßenich, O., et al.: *Renaming and shifted code in structured merging: Looking ahead for precision and performance.* 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2017. **https://doi.org/10.1109/ASE.2017.8115665**

[29] Brun, Y., Holmes, R., et al.: *Proactive detection of collaboration conflicts*, in ESEC/FSE '11: Joint Meet. of the Euro. Softw. Eng. Conf. and the Inter. Symp. on the Foundations of Softw. Eng. ACM, pp. 168-178. 2011. **https://doi.org/10.1145/2025113.2025139**