# Turning Merge Conflicts Into Conflict-Induced Variability

Manuel Ohrndorf
University of Bern
Switzerland

Alexander Boll
University of Bern
Switzerland

Roman Bögli
University of Bern
Switzerland

Timo Kehrer
University of Bern
Switzerland

## Abstract

Merging is central to software version control and collaborative work, yet current techniques force developers to resolve conflicts immediately upon each merge attempt, causing constant interruptions and hampering continuous integration. To mitigate, we propose a paradigm shift: merge conflicts shall be no longer treated as obstacles to be eliminated immediately, but as a form of software variability that explicitly captures diverging developer intentions. Such conflict-induced variability defines alternative behaviors that can be explored, analyzed, and resolved upon request, enabling deferred bulk conflict resolution based on the analysis results. With this, we open an avenue of research around a paradigm that shall preserve the practicality of today's merging techniques, while accelerating traditional versioning workflows through increased flexibility and more effective conflict resolution.

## CCS Concepts

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

## Keywords

Software Merging, Merge Conflict Resolution, Merge Conflict Tolerance, Conflict-Induced Variability, Software Product Lines

**ACM Reference Format:**
Manuel Ohrndorf, Alexander Boll, Roman Bögli, and Timo Kehrer. 2026. Turning Merge Conflicts Into Conflict-Induced Variability. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-NIER '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3786582.3786840

## 1 Introduction

Despite the availability of advanced version control systems such as Git [29], merge conflicts arising from concurrent changes to multiple working copies of a development artifact are unavoidable in collaborative software development. Empirical studies indicate that 10–20% of merge attempts result in conflicts, with rates reaching up to 50% in some cases [8, 14]. Merge conflict resolution thus represents a time-consuming and disruptive task prone to errors [21], sometimes taking hours to days of manual work [8, 21].

Given the significance of the problem, ongoing research on software merging exists almost as long as the problem itself [44]. A key limitation of existing approaches is the implicit assumption that conflict detection and conflict resolution are tightly coupled. Given that conflict resolution is far from being fully automated, developers ultimately have to make merge decisions to successfully finish a merge, bothering them in their daily work and hampering continuous integration [8, 21].

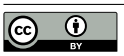
In this paper, we propose a novel paradigm that tolerates merge conflicts temporarily. The key idea is to take a radically different perspective on merge conflicts: rather than enforcing their immediate resolution at every merge attempt, we treat them as a new form of software variability that captures differing developer intentions and makes them explicit. Such *conflict-induced variability* exposes multiple behaviors, which can be systematically explored and analyzed. Based on these analyses, developers can make informed decisions to eventually bind or eliminate conflict-induced variability, thereby shifting from time-consuming resolution of every individual conflict to deferred bulk conflict resolution.

Our approach to realize this research vision is to leverage concepts and techniques from research on software product lines, yet without the need to adopt rigorous product-line development processes. In a nutshell, merge conflicts whose resolution is to be deferred shall be transparently converted to conflict-induced variability, while developers may continue working on individual projections of the integrated version. The binding of accumulated conflict-induced variability, i.e., conflict resolution, shall be informed by adapting concepts from software product-line analysis.

In the sequel, we motivate our novel paradigm through a more detailed analysis of the deficiencies of the state-of-the-art (Sect. 2), present our research goals together with an illustration of our approach (Sect. 3), outline future research steps (Sect. 4), and critically reflect our vision with concluding remarks (Sect. 5).

## 2 State-of-the-Art

**Status quo.** The common way of software merging is three-way-merging, which combines two alternative versions $V_o$ ("ours") and $V_t$ ("theirs") of a development artifact, originating from a common version $V_b$ ("base"), into a merged artifact $V_m$ [11]. For each change in $V_o$ and $V_t$, a merge decision needs to be taken on whether the change shall be incorporated into $V_m$. While certain merge decisions can be taken automatically by consulting the common ancestor, conflicts arise when concurrent changes are incompatible. A multitude of different notions of *incompatible* have been proposed, typically coupled with a concrete technique for conflict detection [30]. Sophisticated techniques work on structural [3, 28] or semantic [33, 38] representations, with the goal of maximizing conflict detection accuracy and minimizing the number of merge conflicts that need to be handled by the developer. On an organizational level, methods were proposed to prevent merge conflicts through early conflict detection and raising conflict-awareness [9, 17]. However, decades of research on conflict detection and prevention have had limited to no impact in practice [18, 34]. Nowadays, mainstream version control systems such as Git uniformly treat all versioned artifacts as




*ICSE-NIER '26, Rio de Janeiro, Brazil*

ACM ISBN 979-8-4007-2425-1/2026/04
https://doi.org/10.1145/3786582.3786840

text, with conflicts being detected as differences in corresponding parts (i.e., lines of text) of $V_o$, $V_t$, and $V_b$. Conflicts are externalized as *conflicting chunks* [8, 21] in a *tentative merge result* supposed to be post-processed by the developer. Interestingly, the recent version control system (VCS) Jujutsu [47] represents conflicts as first-class objects embedded in commits, allowing a version history to advance despite unresolved conflicts. While still in its infancy and far from treating conflicts as analyzable variability, it constitutes a first basic approach by decoupling conflict detection from conflict resolution.

**Machine learning as a game changer?** More recently, the research focus has gradually shifted from conflict detection to conflict resolution. Empirical studies have been conducted to gain deeper insights into the nature of merge conflicts and how developers resolve them in practice, suggesting that the vast majority of chunk resolutions found in practice can be derived from a fixed set of conflict resolution patterns representing rather simple rearrangements of conflicting lines of code [8, 21, 22, 32, 39, 46, 49]. A number of machine learning-based approaches have begun framing merge conflict resolution as a classification problem, where historical conflict resolutions are used as training data for supervised learning [1, 14, 12, 41, 45]. A few papers started to explore GenAI models to generate conflict resolutions beyond a fixed set of resolution patterns [13, 50]. To date, however, the best performing conflict resolution approaches achieve an accuracy of about 75%, yet on narrow benchmark datasets and by offering limited to no insights on the decision-making.

**Motivation of the proposed paradigm shift.** We summarize two major shortcomings that motivate our paradigm shift: (i) Virtually all merge techniques assume a versioning workflow in which conflicts must be resolved as soon as they occur during a merge attempt. Since the accuracy of automated techniques is not yet sufficient to trust fully automated conflict resolutions blindly, developers must therefore continue to review and decide on resolutions themselves. In other words, merely maturing existing techniques within standard versioning workflows is unlikely to eliminate the frequent and time-consuming interruptions caused by merge conflicts; (ii) Conflicts are typically addressed one by one, relying on local syntactic information from the conflicting chunks, or simple high-level characteristics of a merge commit. However, conflicts are not analyzed in a broader context: neither the impact of potential resolutions on program behavior nor the interdependencies between conflicts are taken into account. As a result, consistent decisions across larger sets of conflicting chunks are not supported.

## 3 Research Vision

**Research goals.** To realize our vision that (i) overcomes the *dogma of immediate conflict resolution* and (ii) mitigates the *limitations of isolated conflict resolution*, we devise two research goals:

*RG1*: Instead of enforcing time-consuming conflict resolutions upon each merge attempt, we aim to decouple the integration of changes from conflict resolution by temporarily tolerating conflicts. While the conflicts may be resolved later, continuous development on working copies shall be possible in the meantime.

*RG2*: By centrally accumulating conflict-induced variability, which may be analyzed at the repository site, we aim at synthesizing informed conflict resolution recommendations and guided exploration of resolution spaces, thereby facilitating effective bulk conflict resolution upon request.

**Overview of envisioned workflows.** A high-level overview of how we envision to split the classical workflow of optimistic versioning into two separate workflows is shown in Fig. 1. We focus on the logical steps of *read*, *modify*, *merge* and *write*, which may include minor sub-steps depending on the instantiation of the optimistic versioning model (e.g., *write* = *commit* + *push* in a distributed VCS such as Git). In terms of the *development workflow* (Fig. 1, left), a group of developers (e.g., Alice and Bob) may read a development artifact $V$ from a central repository and concurrently edit the working copies, leading to revisions $V'$ and $V''$, respectively (1). They may *write* their changes to the repository as usual, as long as no conflicts occur (2). As opposed to traditional workflows, writing changes is even possible in case of conflicts, supported by a novel *import* function turning conflicts into *conflict-induced variability* (3). While conflict-induced variability is accumulated centrally in the integrated version $V_\cup$, both Alice and Bob may continue to work on *projections* ($V_{\pi,A}$ and $V_{\pi,B}$), which may be consolidated to expose a minimum of divergences (4). Other developers (e.g., Sally) may *export* and modify a projection that reflects any consolidated state (5). The *conflict resolution workflow* (Fig. 1, right) covers the *merging* step of the traditional workflow. Instead of forcing a conflict resolution upon each write attempt, conflict resolution assistance may be requested at any point in time. In Fig. 1, we illustrate a non-interactive merge approach. Conflict resolution recommendations that bind all conflict-induced variability are generated centrally (1), and developers may pick the most suitable alternative (2).

**Illustration of the approach.** Consider the example in Fig. 2, which shows the evolution of a fragment of a simple graph library written in Java. The base version has been concurrently modified by Alice and Bob. Alice modifies class `Edge` such that edges can be immutable, while Bob adds support for thread-safety through locking. In addition, Alice refactors the method `equals` of class `Edge`, whereas Bob assumes graphs to be directed and therefore also modifies the `equals` method. A merge attempt in a classical VCS yields three conflicts (referred to as $C1$, $C2$, and $C3$ in Fig. 3), refuses the write attempt and asks for a manual conflict resolution.

On the contrary, our solution shall transfer the conflicts into conflict-induced variability. This happens in the background, hidden from the developers, and we may follow the idea of annotation-based software product lines [2] for a suitable internal representation. Accumulated conflict-induced variability may then be resolved at any point in time. In essence, each conflict-induced variation point requires selecting one of its options. Assume that a merge is requested for the integrated repository version shown in Fig. 3. For each of the three conflicts, a canonical resolution [8] would be to accept either Alice's (A) or Bob's (B) modification, constituting a conflict resolution space of $2^3 = 8$ variants (cf. table in Fig. 3). A typical product-line analysis is to check the well-formedness of the derivable variants [42]. One way to do this in a straightforward variant-by-variant fashion is to generate each variant and check whether it compiles. In our example, 4 out of the 8 variants do not compile as the constructor refers to an undeclared instance variable. The remaining 4 variants can run against the test suite, which has been extended by Alice. Note that, contrary to Bob's working copy,

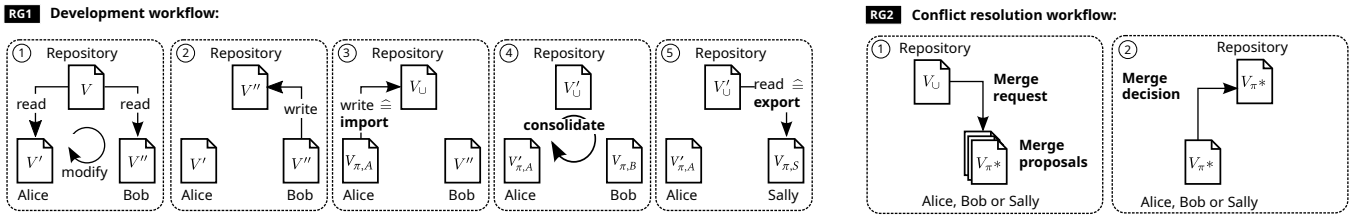



**Figure 1: Traditional workflow of optimistic versioning split into two logically decoupled workflows. The *development workflow* (left) covers the conceptual steps *read*, *modify*, and *write*, whereas *merge* is extracted to the *conflict resolution workflow* (right).**

**Base version**

```
class Edge {
  Node a, b;


  Edge(Node a, Node b) {
    this.a = a; this.b = b;
  }


  boolean equals(Edge e) {
      return a == e.a && b == e.b
          || a == e.b && b == e.a;
  }
}

void testEquals () {
  Node a, b = ...;
  Edge edge = new Edge(a,b);
  assertEquals(edge,edge);
}
```

**Alice's version**

```
class Edge {
  Node a, b;
  boolean immutable;

  Edge(Node a, Node b) {
    this.a = a; this.b = b;
    immutable = false;
  }

  boolean equals(Edge e) {
      return a.equals(e.a) && b.equals(e.b)
          || a.equals(e.b) && b.equals(e.a);
  }
}

void testEquals () {
  // unchanged
  Edge opposite = new Edge(b,a);
  assertEquals(edge,opposite);
}
```

**Bob's version**

```
class Edge {
  Node a, b;
  boolean locked;

  Edge(Node a, Node b) {
    this.a = a; this.b = b;
    locked = false;
  }

  boolean equals(Edge e) {
    return a == e.a && b == e.b;
  }
}


void testEquals () {
  // unchanged
}
```

**Figure 2: Example: A fragment of a simple graph library, concurrently modified by Alice and Bob.**

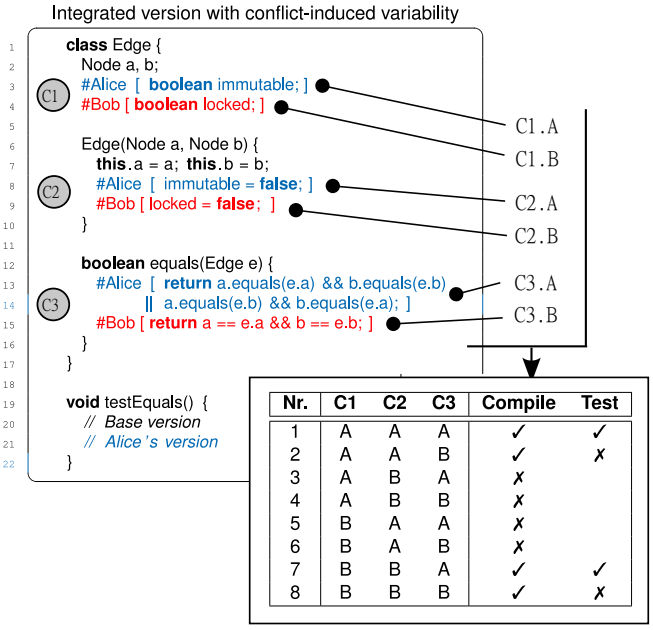



**Figure 3: Sketch of a problem-solution pair: conflict-induced variability (top, left), and recommendation of conflict resolutions (bottom, right).**

the integrated test fails for variants 2 and 8 because Bob's assumption of working with directed graphs does not pass the test. Thus, a reasonable ordering of merge recommendations would rank the remaining two variants 1 and 7 on top. The only decision that needs to be taken is whether we want to support immutable or lockable graphs, leading to consistent merge resolutions for $C1$ and $C2$.

## 4 Future Plans

Our research methodology follows design-science principles [16], where conceptual solutions that generalize beyond the simple example presented in the previous section are realized as research prototypes and empirically evaluated through controlled experiments. Technical challenges shall be addressed iteratively as follows:

**Step 1: Foundations of conflict-induced variability.** In this step, we consider the basic case of turning merge conflicts into conflict-induced variability, assuming that conflicting changes have been applied on a clean base version that does not yet comprise conflict-induced variability. In order to conduct experiments as early as possible, we first aim at a generic solution that reuses mainstream technologies as far as possible. As for conflict detection, a basic implementation shall resort to line-based merging capabilities as offered by, e.g., Git. Conflict markers comprised by a tentative merge shall be transformed into preprocessor macros serving as annotations that capture conflict-induced variability, and projections may be obtained by setting the desired macro variables and

running the preprocessor. Subsequently, we will consider language-specific solutions that exploit the syntactic structure of development artifacts, adopting sophisticated variability mechanisms such as variational abstract syntax trees which facilitate syntactic reasoning over conflict-induced variability in terms of family-based analyses (see steps 3 and 4).

**Step 2: Accumulation of conflict-induced variability.** In this step, we study the case of accumulating conflict-induced variability over a sequence of deferred merge conflict resolutions. While available merge algorithms cannot be applied directly as the latest repository version is typically "polluted" by conflict-induced variability, a 3-way merging scenario shall be established by a novel approach to which we refer as *projectional 3-way merging*, inspired by the notion of projectional editing in filtered software product lines [20, 40]. The idea is to synthesize an "artificial base version" from the latest integrated version by eliminating all conflict-induced variability, using the projection function developed in step 1. A particular challenge is to keep conflict-induced variability concise, which we aim to achieve by consolidating working copies which can be viewed as a generalization of classical merging: Instead of unifying the divergent variants $V_o$ and $V_t$ to a *common* successor version $V_m$ (cf. Sect. 2), each variant gets its *own* successor version $V_o'$ and $V_t'$, and these consolidated versions should have more commonalities than before [35, 36]. Moreover, we anticipate that some conflicts should still be resolved immediately, e.g., those arising from non-local refactorings or other complex restructurings [25, 26] that may generate large amounts of related conflicts [15], where accumulating unresolved conflicts is undesirable.

**Step 3: Non-interactive conflict resolution.** In this step, we follow a non-interactive merging approach by generating conflict resolution recommendations. The goal is to generate a set of variants in which the conflict-induced variability is resolved and which are ranked according to their suitability. Initially, we adopt a variant-by-variant approach as illustrated in Fig. 3: for each distinct combination of conflict-induced variation points, a variant is generated and assessed as a merge candidate. Since recommendations are produced at the repository site, all stages of a CI pipeline can be exploited. We stage static and dynamic analyses sequentially, filtering non-promising variants early with static checks and forwarding only promising ones to more expensive dynamic analyses. Subsequently, we tackle the scalability challenges from the combinatorial explosion of derivable variants, and move beyond the variant-by-variant approach by adopting family-based analyses from software product lines [42]. Building on variability-aware type checking [23], we propose shifting conflict-induced variability binding from compile time to runtime, leveraging variability-aware execution results [24, 35] (e.g., family-based tests) to better evaluate merge candidates.

**Step 4: Interactive exploration of resolution spaces.** Our hypothesis is that we can significantly reduce the number of merge candidates by taking a few merge decisions interactively before switching to non-interactive conflict resolution. The goal is to guide the developer through the merge process by asking for a minimal number of merge decisions while maximizing the reduction of the conflict resolution space. This shall be achieved by propagating merge decisions and by optimizing the conflict resolution order. Complementary, we aim to further reduce interactions by grouping decisions corresponding to changes of transactional development tasks such as bug fixes or feature additions [37], so that a single decision resolves all conflicts in a group.

**Step 5: Empirical evaluation.** As we are interested in how different factors influence the nature of accumulated conflict-induced variability and the effectiveness of semi-automated bulk conflict resolution, we strive for an evaluation strategy that can be applied to early research prototypes from the very beginning, using a controlled experiment setting [48]. To this end, we will simulate the evolution of real-world projects using our methodology. Inspired by recent studies on merge conflict resolution [8, 21], we extract sequences of conflict scenarios from merge commits in open-source repositories. These commits allow us to restore the original situation, replay the evolution with accumulated conflict-induced variability, and use the actual merge commits as ground truth for assessing our resolution strategies. For our simulation-based experiments, we build on the curated project list of Cavalcanti et al. [10] (based on previous work of Munaiah et al. [31] and Beller et al. [5]), which fulfills key requirements such as diversity in size, developer activity, and application domains, as well as the adoption of continuous integration. We refine this dataset by considering historical evolutions rather than isolated merge commits, and extend it with the broader dataset of Boll et al. [8]. A pilot reproducibility study on the extended dataset confirms prior findings that most conflict resolutions can be canonically derived from conflicting chunks, supporting our assumption that conflict-induced variability can be effectively bound in bulk conflict resolution.

## 5 Concluding Remarks

Decades ago, the software engineering community acknowledged that inconsistencies are an inevitable by-product of collaborative development and must be tolerated temporarily rather than immediately eliminated [4, 19]. We argue that merge conflicts deserve the same treatment: instead of enforcing immediate conflict resolution upon each merge attempt, we advocate for conflict tolerance as a means to reduce frequent interruptions and enable deferred and effective bulk conflict resolution upon request. We outlined how such a paradigm shift could be realized by treating merge conflicts as conflict-induced variability, drawing on ideas from software product-line engineering without adopting product-line processes [27, 7]. By turning merge conflicts into conflict-induced variability, we also reveal new opportunities for research by unifying software version and variability management [6, 43].

However, we acknowledge that many questions remain unanswered. From a tooling perspective, it is unclear how the proposed techniques can be realized on top of existing tools and integrated into the envisioned workflows, and controlled experiments alone will not suffice to fully assess all implications of the proposed paradigm shift. Ultimately, field studies with more mature prototypes will be required to evaluate potential negative effects and organizational consequences. Our overarching goal is to understand when deferring conflict resolution is beneficial and when immediate resolution is the preferred strategy. To this end, our approach also allows developers to fall back to conventional merging and directly resolve conflicts as they occur.

## References


[1] W. Aldndni, N. Meng, and F. Servant. 2023. Automatic prediction of developers' resolutions for software merge conflicts. *Journal of Systems and Software*, 206.

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer.

[3] Sven Apel, Olaf Leßenich, and Christian Lengauer. 2012. Structured merge with auto-tuning: balancing precision and performance. In *Proc. Int'l Conf. on Automated Software Engineering*, 120–129.

[4] Robert Balzer. 1991. Tolerating inconsistency (software development). In *Proc. Int'l Conf. on Software Engineering*, 158–159.

[5] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, my tests broke the build: an explorative analysis of travis ci with github. In *Proc. Int'l Working Conf. on Mining Software Repositories*. IEEE, 356–367.

[6] Thorsten Berger, Marsha Chechik, Timo Kehrer, and Manuel Wimmer. 2019. Software evolution in time and space: unifying version and variability management (dagstuhl seminar 19191). *Dagstuhl Reports*, 9, 5, 1–30.

[7] Roman Bögli, Alexander Boll, Alexander Schultheiß, and Timo Kehrer. 2025. Beyond software families: community-driven variability. In *Proc. Int'l Conf. on the Foundations of Software Engineering: Ideas, Visions and Reflections*, 571–575.

[8] Alexander Boll, Yael Van Dok, Manuel Ohrndorf, Alexander Schultheiß, and Timo Kehrer. 2024. Towards semi-automated merge conflict resolution: is it easier than we expected? In *Proc. Int'l Conf. on Evaluation and Assessment in Software Engineering*, 282–292.

[9] Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. 2013. Early detection of collaboration conflicts and risks. *IEEE Trans. on Software Engineering*, 39, 10, 1358–1375.

[10] Guilherme Cavalcanti, Paulo Borba, Georg Seibt, and Sven Apel. 2019. The impact of structure on software merging: semistructured versus structured merge. In *Proc. Int'l Conf. on Automated Software Engineering*, 1002–1013.

[11] Reidar Conradi and Bernhard Westfechtel. 1998. Version Models for Software Configuration Management. *ACM Computing Surveys*, 30, 2, 232–282.

[12] Elizabeth Dinella, Todd Mytkowicz, Alexey Svyatkovskiy, Christian Bird, Mayur Naik, and Shuvendu Lahiri. 2023. DeepMerge: Learning to Merge Programs. *IEEE Trans. on Software Engineering*, 49, 4, (Apr. 2023), 1599–1614. Retrieved Feb. 14, 2025 from.

[13] Jinhao Dong, Qihao Zhu, Zeyu Sun, Yiling Lou, and Dan Hao. 2023. Merge Conflict Resolution: Classification or Generation? In *Proc. Int'l Conf. on Automated Software Engineering*, 1652–1663. Retrieved Feb. 14, 2025 from.

[14] Paulo Elias, Heleno De S. Campos, Eduardo Ogasawara, and Leonardo Gresta Paulino Murta. 2023. Towards accurate recommendations of merge conflicts resolution strategies. *Information and Software Technology*, 164, 107332.

[15] Max Ellis, Sarah Nadi, and Danny Dig. 2022. Operation-based refactoring-aware merging: an empirical evaluation. *IEEE Trans. on Software Engineering*, 49, 4, 2698–2721.

[16] Emelie Engström, Margaret-Anne Storey, Per Runeson, Martin Höst, and Maria Teresa Baldassarre. 2020. How software engineering research aligns with design science: a review. *Empirical Software Engineering*, 25, 2630–2660.

[17] H Christian Estler, Martin Nordio, Carlo A Furia, and Bertrand Meyer. 2014. Awareness and merge conflicts in distributed software development. In *Proc. Int'l Conf. on Global Software Engineering*, 26–35.

[18] Jacky Estublier, David Leblang, André van der Hoek, Reidar Conradi, Geoffrey Clemm, Walter Tichy, and Darcy Wiborg-Weber. 2005. Impact of software engineering research on the practice of software configuration management. *ACM Trans. on Software Engineering and Methodology*, 14, 4, 383–430.

[19] Anthony CW Finkelstein, Dov Gabbay, Anthony Hunter, Jeff Kramer, and Bashar Nuseibeh. 1994. Inconsistency handling in multiperspective specifications. *IEEE Trans. on Software Engineering*, 20, 8, 569–578.

[20] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2015. The ECCO Tool: Extraction and Composition for Clone-and-Own. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 665–668.

[21] Gleiph Ghiotto, Leonardo Murta, Márcio Barros, and André van der Hoek. 2020. On the Nature of Merge Conflicts: A Study of 2,731 Open Source Java Projects Hosted by GitHub. *IEEE Trans. on Software Engineering*, 46, 8, 892–915.

[22] Heleno de S. Campos Junior, Gleiph Ghiotto L. de Menezes, Márcio de Oliveira Barros, André van der Hoek, and Leonardo Gresta Paulino Murta. 2024. How code composition strategies affect merge conflict resolution? *Journal of Software Engineering Research and Development*, 12, 1.

[23] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. 2012. Type Checking Annotation-Based Product Lines. *ACM Trans. on Software Engineering and Methodology*, 21, 3, 14:1–14:39.

[24] Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. 2012. Toward Variability-Aware Testing. In *Proc. Int'l Workshop on Feature-Oriented Software Development*, 1–8.

[25] Timo Kehrer. 2015. *Calculation and propagation of model changes based on user-level edit operations: a foundation for version and variant management in model-driven engineering*. Ph.D. Dissertation. University of Siegen.

[26] Timo Kehrer, Abdullah Alshanqiti, and Reiko Heckel. 2017. Automatic inference of rule-based specifications of complex in-place model transformations. In *Proc. Int'l Conf. on Theory and Practice of Model Transformations*, 92–107.

[27] Timo Kehrer, Thomas Thüm, Alexander Schultheiß, and Paul Maximilian Bittner. 2021. Bridging the gap between clone-and-own and software product lines. In *Proc. Int'l Conf. on Software Engineering: New Ideas and Emerging Results*, 21–25.

[28] Olaf Leßenich, Sven Apel, and Christian Lengauer. 2015. Balancing precision and performance in structured merge. *Proc. Int'l Conf. on Automated Software Engineering*, 22, 3, 367–397.

[29] Jon Loeliger and Matthew McCullough. 2012. *Version Control with Git: Powerful tools and techniques for collaborative software development*. O'Reilly Media, Inc.

[30] Tom Mens. 2002. A state-of-the-art survey on software merging. *IEEE Trans. on Software Engineering*, 28, 5, 449–462.

[31] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating github for engineered software projects. *Empirical Software Engineering*, 22, 3219–3253.

[32] Hoai Le Nguyen and Claudia-Lavinia Ignat. 2018. An analysis of merge conflicts and resolutions in git-based open source projects. *Computer Supported Cooperative Work*, 27, 741–765.

[33] Hung Viet Nguyen, My Huu Nguyen, Son Cuu Dang, Christian Kästner, and Tien N Nguyen. 2015. Detecting semantic merge conflicts with variability-aware execution. In *Proc. Int'l Symposium on Foundations of Software Engineering (FSE)*, 926–929.

[34] Stack Overflow. 2022. Stack overflow developer survey 2022 - version control systems. Accessed on 25 Sep 2025. Retrieved Feb. 28, 2025 from https://survey.stackoverflow.co/2022/#section-version-control-version-control-systems.

[35] Dennis Reuling, Udo Kelter, Johannes Bürdek, and Malte Lochau. 2019. Automated n-way program merging for facilitating family-based analyses of variant-rich software. *ACM Trans. on Software Engineering and Methodology*, 28, 3, 1–59.

[36] Dennis Reuling, Malte Lochau, and Udo Kelter. 2019. From imprecise n-way model matching to precise n-way model merging. *J. Object Technol.*, 18, 2, 8–1.

[37] Maik Schmidt, Sven Wenzel, Timo Kehrer, and Udo Kelter. 2009. History-based merging of models. In *Proc. Int'l Workshop on Comparison and Versioning of Software Models*, 13–18.

[38] Danhua Shao, Sarfraz Khurshid, and Dewayne E Perry. 2009. SCA: a semantic conflict analyzer for parallel changes. In *Proc. Int'l Symposium on Foundations of Software Engineering (FSE)*, 291–292.

[39] Bowen Shen, Cihan Xiao, Na Meng, and Fei He. 2021. Automatic detection and resolution of software merge conflicts: are we there yet? *arXiv preprint arXiv:2102.11307*.

[40] Stefan Stănciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wąsowski. 2016. Concepts, Operations, and Feasibility of a Projection-Based Variation Control System. In *Proc. Int'l Conf. on Software Maintenance and Evolution*. IEEE, 323–333.

[41] Alexey Svyatkovskiy, Sarah Fakhoury, Negar Ghorbani, Todd Mytkowicz, Elizabeth Dinella, Christian Bird, Jinu Jang, Neel Sundaresan, and Shuvendu K. Lahiri. 2022. Program merge conflict resolution via neural transformers. In *Proc. Int'l Symposium on Foundations of Software Engineering (FSE)*, 822–833.

[42] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys*, 47, 1, 6:1–6:45.

[43] Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. 2019. Towards efficient analysis of variation in time and space. In *Proc. Int'l Systems and Software Product Line Conference*, 57–64.

[44] Walter F. Tichy. 1982. Design, Implementation, and Evaluation of a Revision Control System. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, 58–67.

[45] Marina Bianca Trif and Radu Razvan Slavescu. 2021. Towards Predicting Merge Conflicts in Software Development Environments. In *Proc. Int'l Conf. on Intelligent Computer Communication and Processing (ICCP)*, 251–256.

[46] Gustavo Vale, Claus Hunsen, Eduardo Figueiredo, and Sven Apel. 2022. Challenges of Resolving Merge Conflicts: A Mining and Survey Study. *IEEE Trans. on Software Engineering*, 48, 12, 4964–4985.

[47] Martin von Zweigbergk. 2026. Jujutsu - version control systems. Accessed on 07 Jan 2026. https://jj-vcs.dev/.

[48] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[49] Ryohei Yuzuki, Hideaki Hata, and Kenichi Matsumoto. 2015. How we resolve conflict: an empirical study of method-level conflict resolution. In *Proc. Int'l Workshop on Software Analytics*. IEEE, 21–24.

[50] Jialu Zhang, Todd Mytkowicz, Mike Kaufman, Ruzica Piskac, and Shuvendu K Lahiri. 2022. Using pre-trained language models to resolve textual and semantic merge conflicts. In *Proc. Int'l Symp. on Software Testing and Analysis*, 77–88.