This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3538326

A Retrospective of ChangeDistiller: Tree Differencing for Fine-Grained Source Code Change Extraction

Beat Fluri, Michael Würsch, Martin Pinzger, and Harald Gall

***Abstract*—In the early development of source code change analysis, methodologies primarily relied on simple textual differencing, which treated code as mere text and identified changes through lines that were added, modified, or deleted. This approach overlooked the rich semantic information embedded within the code, highlighting significant limitations in textual analysis and differencing that required a more precise and language-aware foundation. Our research on ChangeDistiller pioneered the use of abstract syntax trees and associated tree edits for change analysis. We were among the first to introduce a tree-differencing algorithm for source code, enabling a fine-grained examination of modifications. ChangeDistiller has since been widely adopted by researchers in the field of mining software repositories. This paper reflects on the evolution of our technique, its influence on subsequent research, and its role in the advancement of change analysis methodologies. In addition, we explore how contemporary techniques and tools can draw on our foundational work to enhance their effectiveness.**

***Index Terms*—source code change analysis, change types, tree edits, mining software repositories**

I. The road to ChangeDistiller

CHANGE is the principle at the core of nature. Evolution research seeks to understand the processes and mechanisms that drive change in biological, cultural, and technological systems over time. The insights gained from this research elucidate the diversity and complexity of these systems, facilitate better ecosystem management and sustainable development, and empower us to create predictive models that guide interventions. This understanding is relevant in both biological and software systems.

With the emergence of open-source software repositories around the turn of the millennium, including SourceForge and GitHub, researchers gained unprecedented access to extensive source code and its change history. This advancement contributed to a new research area within software maintenance and evolution: Mining Software Repositories (MSR). We were among the first to investigate large software repositories by means of data mining and machine learning. We learned about various relations between code elements, software developers and teams, defect dependencies, or bug fixing effects. With our techniques, we leveraged repository data to identify foundational principles of software evolution: which code elements change how, when, or by whom, with the ultimate goal to better support developers in the creation and maintenance of large-scale complex software systems.

A key challenge for MSR researchers was the identification and classification of changes between successive versions of a program. We found that research was often hindered by the low quality of available information, as changes were typically tracked at a textual level—simply identifying lines added or deleted. This approach lacked the resolution, syntax, and semantics necessary for drawing meaningful conclusions. More advanced methods attempted to identify method-level changes through syntactical analysis but they still struggled to accurately classify those changes—such as the addition of method invocations in the else branch of an if-statement. ChangeDistiller [1] addressed this gap by employing abstract syntax trees (ASTs) to represent two versions of a program, along with a tree-differencing algorithm to identify the changes between them. The algorithm generated an edit script comprised of basic tree edit operations, which, when applied, transformed the original tree into the modified one. These operations were then categorized and included in our comprehensive taxonomy of source code changes, allowing us to analyze the syntactic nature of code changes for the first time ever.

II. ChangeDistiller

At the core of ChangeDistiller lays the *change distilling* algorithm. It defines, classifies, and analyzes fine-grained source code changes. Change distilling provides a *taxonomy of source code changes* that precisely defines *change types* according to tree edit operations in the abstract syntax tree (AST). We use the basic tree edit operations *insert*, *delete*, *move*, and *update* applicable to AST nodes. In addition, the taxonomy classifies each change type according to a *change significance level* scheme. This level expresses the possible impact a change type may have on other source code entities and whether it may be functionality-preserving or functionality-modifying.

As a result, the taxonomy defines 40 change types for source code entity types that are defined in a programming language and representable in an AST. The change types are divided into *body* and *declaration* part categories of attributes, classes, and methods. Each change type obtains a change significance level of *none*, *low*, *medium*, *high*, or *crucial*. For certain change types the change significance level is adapted to the accessibility modifier of a source code entity. For instance, a return type change of a public method has a higher change significance level than of a private method.

We use the taxonomy to extract fine-grained source code changes. Our change distilling algorithm uses tree-differencing on subsequent AST versions of an object-oriented class. The

Authorized licensed use limited to: Universitaet Klagenfurt. Downloaded on March 11,2025 at 15:04:18 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3538326

algorithm calculates an edit script that contains basic tree edit operations and transforms the older into the newer AST. That means, the edit script contains exactly those source code changes that were applied to the class between the two versions. The change distilling algorithm is based on the generic tree-differencing algorithm presented by Chawathe *et al.* [2] that we customize to be applicable on pairs of ASTs. ChangeDistiller is the implementation of the change distilling algorithm for the Java programming language.

Leveraging the information provided by ASTs permits us to get precise information about a source code change. In addition to the information that a particular source code entity has changed, tree edit operations also provide information about the location of the change. For instance, we can tell that the method invocation statement `foo.bar()` was moved from the then-part to the else-part of the if-statement that has the condition `foo == null`.

We evaluated ChangeDistiller in three studies:

1) Qualifying change coupling: In a first study, we applied our change taxonomy and found that in many cases a large number of added and/or deleted lines did not show significant changes, but small textual adaptations (such as indentation, etc.). [3]. Our approach allowed us to relate all change couplings to the significance of the identified change types. As a result, change couplings between code entities can be qualified and less relevant couplings can be filtered out.

2) Benchmark for change extraction: In the original paper we used a benchmark comprising 1,064 manually classified changes across 219 revisions of eight methods from three open-source projects [1]. The evaluation relied on metrics such as mean absolute error and mean absolute percentage error to measure the algorithm's accuracy in identifying changes. The improved algorithm outperformed the baseline by significantly reducing the percentage error from 79% to 34%, demonstrating its capability to approximate the minimum conforming edit script more closely. This success was attributed to enhancements like the use of bigram similarity for matching, a best match algorithm for nodes, and dynamic thresholds for subtree matching. Overall, the evaluation confirmed that the new approach offers a substantial improvement in detecting and classifying source code changes compared to existing methods.

3) Co-evolution of code and comments: The research on ChangeDistiller has served as a cornerstone for numerous empirical studies, including an analysis by Fluri *et al.* on the co-evolution of source code and comments [4], [5]. Source code comments play a crucial role in preserving design decisions and conveying the intent of the code to developers and maintainers. However, a common belief suggests that developers frequently neglect documenting their code and fail to keep comments up-to-date. Our study aimed to address this issue by investigating: *To what extent do developers keep code and comments in sync?* We utilized ChangeDistiller in combination with heuristics based on proximity and textual similarity to map comments to specific code entities. An analysis of eight open- and closed-source systems revealed that nearly half of the code changes were not accompanied by corresponding updates to comments. However, when comments were present, they typically evolved in parallel with the associated code, albeit often with a delay of one or more revisions, particularly during phases of consolidation.

III. The impact of ChangeDistiller

The availability of ChangeDistiller and its detailed insights into source code changes have paved the way for extensive research opportunities. We reflect on the impact on our work and explore how others have utilized and extended ChangeDistiller in various directions.

A. Impact on our own research

Next, we highlight several exemplary cases of our research involving ChangeDistiller and its impact:

1) Ontologies in software engineering: Through SEON [6], an open-source ontology of key concepts in software engineering, ChangeDistiller found wider applications in other research projects. SEON formally describes the ChangeDistiller change types using the Web Ontology Language OWL and enables reasoning about changes and the integration of these concepts into research tools in a unified way.

2) Search-driven software engineering: One notable application was our query framework for software evolution data, introduced by Würsch *et al.* in [7]. The approach envisioned developer-centric tooling—similar to GitHub Copilot nowadays but with a stronger focus on answering questions about code base evolution to improve code comprehension rather than generating code. A subsequent user study demonstrated that participants could answer common developer questions with greater accuracy in less time using this framework [8].

3) Software analysis as a Service: Building on that, Ghezzi *et al.* introduced SOFAS (Sofware Analysis as a Service), a RESTful architecture providing a simple yet effective way to offer software analyses as a service [9]. SOFAS allowed researchers and practitioners to incorporate the ChangeDistiller algorithm into more complex analysis workflows. The data consumed and produced by the individual services were defined and represented using the SEON ontology, ensuring consistency and interoperability across analysis steps.

4) Bug prediction: Giger *et al.* used fine-grained code change information to predict bugs [10] and their fix times [11]. They compared different prediction models—with and without fine-grained code changes—and the results showed that fine-grained code change information can significantly improve the performance of the models.

5) Beyond source code changes: Pinzger *et al.* extended ChangeDistiller to extract detailed information on changes from other information sources, such as web service interfaces [12], feature models [13] and build files [14]. This allowed them to study the evolution of the Linux kernel features [13] and the nature of build changes in open-source projects [15]. They also investigated ways to improve AST differencing and presented IJM [16]. The results of an empirical study showed that IJM provides significantly better accuracy for move and update actions. IJM was integrated into DiffViz [17], a tool to facilitate the navigation of code changes.

Authorized licensed use limited to: Universitaet Klagenfurt. Downloaded on March 11,2025 at 15:04:18 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3538326

6) Semantics of source code changes: While ChangeDistiller highlights syntactic differences between two versions of a program, it is not capable of explaining changes in semantics. Glock *et al.* developed PASDA [18] that uses differential symbolic execution to prove non-/equivalence of program pairs based on path level. These proofs benefit use cases such as test case prioritization, fault localization, and debugging.

B. Impact on other research

Several other researchers built on ChangeDistiller to study the evolution of software systems, to better support developers, and to develop more accurate change extraction approaches.

1) Empirical studies: Soetens *et al.* [19] explored the trade-off between test set reduction and the fault detection ability of two test selection heuristics based on fine-grained code changes extracted with ChangeDistiller. Raemaekers *et al.* [20] used ChangeDistiller to measure the size of the changed functionality between major, minor, and patch releases to reason about breaking changes in the Maven repository. Zhong and Meng [21] computed delta graphs for related changes with the help of ChangeDistiller. These graphs were then used to analyze the information overlap between past and new bug fixes, showing that up to 41.3% of the new bug fixes can be derived from previous fixes. Their work highlights the potential of fine-grained code changes for the automation of software engineering tasks such as automatic program repair (APR), which has been a hot research topic in the last decade.

2) Developer support: ChangeDistiller also made an impact on research in the active fields of program comprehension and developer recommendation systems. Kawrykow and Robillard [22] enhanced the output generated by ChangeDistiller with Partial Program Analysis (PPA) to further distinguish important changes from non-essential modifications, such as local variable refactorings, or textual differences induced as part of a rename refactoring. Rubin and Chechik [23] focus on feature location for families of related software products realized via code cloning. The authors found ChangeDistiller useful to automate identification of regions of code that likely implement a feature of interest by comparing the analyzed program variant to an earlier release which does not contain the feature. Holmes and Walker used ChangeDistiller to develop YooHoo [24], an approach to proactively notify developers about changes in libraries they depend on. Catolino *et al.* enhanced change prediction models using developer-related factors and made use of ChangeDistiller to evaluate the performance of their models [25]. Kreuzer *et al.* [26] used a modified version of ChangeDistiller and syntactical similarity metrics in combination with two self-tuning clustering algorithms to detect groups of similar code changes. Tsantalis *et al.* [27] presented RefactoringMiner, an approach to detect refactorings in the commit history of software projects. Janke and Mäder presented an approach that uses frequent subgraph mining to identify change patterns from AST edits [28]. Their method relied on GumTree, which is based on ChangeDistiller and will be covered in the next section. An evaluation of six selected patterns with 31 participants showed that the patterns are relevant for programming activities, and IDEs should offer automated support to perform such changes.

3) Improving change detection accuracy: Several researchers succeeded in improving the change detection accuracy of ChangeDistiller. Dotzler *et al.* presented MTDiff [29], an approach that provides several optimizations, in particular to the mapping phase, to better map moved nodes. They also implemented optimizations for ChangeDistiller and showed that they can produce shorter edit scripts for most of the analyzed files than existing approaches. The most recent advancements in tree-differencing were presented by Falleri *et al.* [30], [31] and Alikhanifard *et al.* [32]: In contrast to ChangeDistiller, GumTree considers the full AST of a source file and also computes the differences between single statements. Furthermore, it improves the mapping of tree nodes by combining a greedy top-down algorithm to find isomorphic subtrees with a bottom-up algorithm to map parents and their descendant nodes. This results in better and more concise edit scripts. Falleri *et al.* later introduced an algorithm to improve the final mapping phase of GumTree, which results in shorter edit scripts and shorter matching time [31]. Because of its advanced node mapping approach, GumTree became the most widely used approach for extracting fine-grained code changes. Alikhanifard *et al.* discuss five limitations of existing AST differencing tools and present RefactoringMiner 3.0 to resolve them [32]. A comparison with five state-of-the-art tools shows that RefactoringMiner 3.0 outperforms them in terms of precision and recall significantly down to statement level. It also generates a perfect diff for 87.9% of the commits, which is more than 20% higher than the ratio of existing approaches. This makes RefactoringMiner 3.0 the best tree-differencing approach that is currently available.

IV. The road ahead

In this section, we envision future research directions inspired by the capabilities of ChangeDistiller and the advancements made through its use and refinement.

A. AST differencing

Several approaches have been proposed to enhance ChangeDistiller, particularly in matching tree nodes. The authors of GumTree suggested that further improvements could be made by incorporating hyperparameter tuning, as demonstrated by Martinez *et al.* [33]. Alikhanifard *et al.* acknowledged limitations in their approach, such as inaccuracies in mapping test code. However, with an overall precision and recall exceeding 99% and a perfect diff rate of 88.5%, achieving significant improvements is challenging. Instead, we propose extending RefactoringMiner to support additional programming languages beyond Java and Python, unlocking a broader range of data to derive valuable insights.

B. Higher-level code changes

Besides having accurate information about code changes, developers need assistance in understanding them. Tree-edit operations represent code changes accurately but they are difficult for developers to interpret. The ChangeDistiller taxonomy of code changes groups tree-edit operations into meaningful

Authorized licensed use limited to: Universitaet Klagenfurt. Downloaded on March 11,2025 at 15:04:18 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3538326

changes, however, it does not consider code changes that span multiple source files or refactorings. ClDiff [34] groups tree-edit operations and links them according to five predefined links that can be interpreted as five different change types. RefactoringMiner [32] extends existing approaches by grouping tree-edit operations to 102 refactorings. Future work should focus on studying their usefulness in understanding code changes and on adding missing change types.

C. Visualization of code changes

Another potential for improvement lies in the representation of code changes. Current diff-tools either show a unified line-based diff or a side-by-side comparison of two versions of a source file. They do not provide any explanation of the code changes and the intent behind them. As suggested by Alikhanifard *et al.* future diff visualization could provide a summary or explanation generated from various sources, for example, code changes, commit messages, linked issues, or discussions from the corresponding pull request to shine light on the intention behind it [32]. Summarization could be accomplished using large language models (LLMs). Additionally, future approaches could integrate the information on semantic differences similar to PASDA [18] to help developers understand the changes in the semantics of programs.

D. Change impact analysis

A key question in understanding code changes concerns their impact on other code entities and modules, for instance, to detect undesirable side effects/faults introduced by a change. Most existing approaches use program dependency graphs, code similarity, or past change couplings for assessing the impact of code changes. However, they have shown to generate large impact sets with low precision. Athena, a recent approach by Yan *et al.* [35] combines dependency graphs with a deep code representation to output methods that are most likely impacted by a code change. While Athena outperforms state-of-the-art impact analysis approaches, it still suffers from several shortcomings. First, the approach is based on method-level and does not show the impact of code changes on the statement level, which is, in our opinion, still too coarse-grained. Second, while it uses the term ”semantics” it mainly means textual similarity—it does not detect or explain any changes in behavior of statements and methods. In this direction, Gyori *et al.* [36] and Hanam *et al.* and [37] presented two approaches to compute the semantic impact of a code change by using symbolic execution or abstract interpretation. These approaches will benefit from more accurate code diffs and future work should focus on investigating this potential.

E. Commit message generation

Commit messages describe code changes in natural language. Several approaches train models that generate a corresponding commit message from a set of code changes. Buse *et al.* [38] used predefined rules, while more recent research investigated models trained with deep learning to generate commit messages [39], [40]. For instance, Tao *et al.* [40] introduced KADEL, an approach that fine-tunes the CodeT5 model [41] with pairs of code changes and commit messages from the MCMD dataset [42] considering five programming languages. The results show that KADEL outperforms state-of-the-art approaches. Furthermore, a comparison of KADEL with the much larger ChatGPT (3.5-turbo) shows that both models perform equally well. The comparison with more recent LLMs, such as ChatGPT 4o, Claude 3.5, Code Llama, or Gemini, is suggested for future work. Furthermore, all these approaches might benefit from model fine-tuning based on detailed information about code changes extracted with tree-differencing.

F. Automated program repair

Typical APR approaches use machine or deep learning to train or fine-tune a model from pairs of buggy code and the corresponding fixed code. For instance, Jiang *et al.* [43] evaluated ten code language models (CLMs), such as CodeT5 and inCoder [44], on four Java benchmarks and showed that their fine-tuned models fix significantly more bugs than other approaches. Xia and Zhang [45] proposed ChatRepair, the first fully automated conversation-driven APR approach that uses ChatGPT 3.5-turbo. The results show that ChatRepair could correctly fix 162 out of 337 bugs for $0.42 each which is even more than the fine-tuned CLMs could fix. Recently, Hidvégi *et al.* [46] proposed a similar approach called CigaR with the goal of repairing more bugs while minimizing token costs. CigaR also uses ChatGPT 3.5-turbo but improves the prompting strategies through iterative prompting, search rebooting and patch multiplication. The evaluation shows that CigaR managed to reduce the token cost by 73% while fixing more bugs than ChatRepair. The prompts used by existing approaches typically contain an example of a bug and some bug details (*e.g.*, the results from a failed test). For future work and similar to the commit message generation, it would be interesting to investigate whether the detailed information about bug fixes extracted with tree-differencing could help improve existing APR approaches.

G. Automated refactoring

Modernizing large code bases remains a common challenge in the software industry. Tasks such as framework migrations can require a prohibitive amount of manual effort, as one of the authors experienced firsthand during a recent migration from Java EE to Jakarta EE. Auto-refactoring engines such as OpenRewrite,[1] significantly reduce the time and effort required for such tasks, transforming what could take hours into mere minutes. OpenRewrite, to the best of our knowledge, is not directly related to or influenced by ChangeDistiller. However, it uses a specialized form of ASTs called *lossless semantic trees* (LSTs) to represent source code. OpenRewrite utilizes *recipes*—collections of predefined search and refactoring operations—that operate on LSTs to perform automated code transformations. In this sense, OpenRewrite recipes bear some similarity to ChangeDistiller's edit scripts,

[1] https://docs.openrewrite.org, visited January 10, 2025

Authorized licensed use limited to: Universitaet Klagenfurt. Downloaded on March 11,2025 at 15:04:18 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3538326

which describe changes between versions of a codebase. Currently, OpenRewrite recipes must be explicitly authored by a developer. Combining a change detection algorithm akin to ChangeDistiller with an auto-refactoring engine like OpenRewrite presents an intriguing possibility: mining large-scale source code repositories for common refactorings and automatically generating recipes to apply these transformations across other code bases. Alternatively, these recipes might also be learned by LLMs fine-tuned with examples from past migrations.

V. Conclusions

Analyzing software evolution requires the identification of specific code changes that occur between program versions. With ChangeDistiller, we introduced the first technique that goes beyond textual diffs of programs and can compute a fine-grained code change history through AST differentiation. Our approach and tool have been widely used in studies of open-source software development, from advancements in code analysis to change patterns identification, visualizations, empirical studies, and specialized tools. ChangeDistiller and our closely related works have accumulated more than 2,500 citations and cover a period from 2007 to the present day. Today, machine learning techniques and in particular LLMs offer new insights into the development of source code. However, a nuanced understanding of change types and classifications provided by tree-differencing approaches, such as ChangeDistiller, remains essential for programmatic comprehension of code changes.

References

[1] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall, "Change Distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[2] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," in *Proceedings of the International Conference on Management of Data*. ACM, 1996, pp. 493–504.

[3] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," in *Proceedings of the International Conference on Program Comprehension*. IEEE, 2006, pp. 35–45.

[4] B. Fluri, M. Würsch, and H. C. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *Proceedings of the Working Conference on Reverse Engineering*. IEEE, 2007, pp. 70–79.

[5] B. Fluri, M. Würsch, E. Giger, and H. C. Gall, "Analyzing the co-evolution of comments and source code," *Software Quality Journal*, vol. 17, no. 4, pp. 367–394, December 2009.

[6] M. Würsch, G. Ghezzi, M. Hert, G. Reif, and H. Gall, "Seon: A pyramid of ontologies for software evolution and its applications," *Computing*, vol. 94, no. 11, pp. 857–885, 2012.

[7] M. Würsch, G. Ghezzi, G. Reif, and H. C. Gall, "Supporting developers with natural language queries," in *32nd Proceedings of the International Conference on Software Engineering*. ACM, 2010.

[8] M. Würsch, E. Giger, and H. Gall, "Evaluating a query framework for software evolution data," *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 4, pp. 38–38, 2013.

[9] G. Ghezzi and H. C. Gall, "Sofas: A lightweight architecture for software analysis as a service," in *Proceedings of the Working Conference on Software Architecture*. IEEE, 2011, pp. 93–102.

[10] E. Giger, M. Pinzger, and H. Gall, "Comparing fine-grained source code changes and code churn for bug prediction," in *Proceedings of the Working Conference on Mining Software Repositories*. ACM, 2011, pp. 83–92.

[11] ——, "Predicting the fix time of bugs," in *Proceedings of the International Workshop on Recommendation Systems for Software Engineering*. ACM, 2010, pp. 52–56.

[12] D. Romano and M. Pinzger, "Analyzing the evolution of web services using fine-grained changes," in *Proceedings of the International Conference on Web Services*. IEEE, 2012, pp. 392–399.

[13] N. Dintzner, A. Deursen, and M. Pinzger, "Analysing the Linux kernel feature model changes using FMDiff," *Software and Systems Modeling*, vol. 16, no. 1, pp. 55–76, 2017.

[14] C. Macho, S. Mcintosh, and M. Pinzger, "Extracting build changes with BuildDiff," in *Proceedings of the International Conference on Mining Software Repositories*. IEEE, 2017, pp. 368–378.

[15] C. Macho, S. Beyer, S. McIntosh, and M. Pinzger, "The nature of build changes," *Empirical Software Engineering*, vol. 26, no. 3, p. 32, 2021.

[16] V. Frick, T. Grassauer, F. Beck, and M. Pinzger, "Generating accurate and compact edit scripts using tree differencing," in *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE, 2018, pp. 264–274.

[17] V. Frick, C. Wedenig, and M. Pinzger, "DiffViz: A diff algorithm independent visualization tool for edit scripts," in *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE, 2018, pp. 705–709.

[18] J. Glock, J. Pichler, and M. Pinzger, "Pasda: A partition-based semantic differencing approach with best effort classification of undecided cases," *Journal of Systems and Software*, vol. 213, p. 112037, 2024.

[19] Q. D. Soetens, S. Demeyer, A. Zaidman, and J. Pérez, "Change-based test selection: an empirical evaluation," *Empirical Software Engineering*, vol. 21, no. 5, pp. 1990–2032, 2016.

[20] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning and impact of breaking changes in the maven repository," *Journal of Systems and Software*, vol. 129, pp. 140–158, 2017.

[21] H. Zhong and N. Meng, "Towards reusing hints from past fixes," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2521–2549, 2018.

[22] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *Proceedings of the International Conference on Software Engineering*. ACM, 2011, pp. 351–360.

[23] J. Rubin and M. Chechik, "Locating distinguishing features using diff sets," in *Proceedings of the International Conference on Automated Software Engineering*. ACM, 2012, pp. 242–245.

[24] R. Holmes and R. J. Walker, "Customized awareness: recommending relevant external change events," in *Proceedings of the International Conference on Software Engineering*. ACM, 2010, pp. 465–474.

[25] G. Catolino, F. Palomba, A. De Lucia, F. Ferrucci, and A. Zaidman, "Enhancing change prediction models using developer-related factors," *Journal of Systems and Software*, vol. 143, pp. 14–28, 2018.

[26] P. Kreutzer, G. Dotzler, M. Ring, B. M. Eskofier, and M. Philippsen, "Automatic clustering of code changes," in *Proceedings of the International Conference on Mining Software Repositories*. ACM, 2016, p. 61–72.

[27] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the International Conference on Software Engineering*. IEEE, 2018, pp. 483–494.

[28] M. Janke and P. Mäder, "$fs^3{}_{change}$: A scalable method for change pattern mining," *IEEE Transactions on Software Engineering*, vol. 49, no. 6, pp. 3616–3629, 2023.

[29] G. Dotzler and M. Philippsen, "Move-optimized source code tree differencing," in *Proceedings of the International Conference on Automated Software Engineering*. ACM, 2016, pp. 660–671.

[30] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the International Conference on Automated Software Engineering*. ACM, 2014, pp. 313–324.

[31] J.-R. Falleri and M. Martinez, "Fine-grained, accurate and scalable source differencing," in *Proceedings of the International Conference on Software Engineering*. ACM, 2024, pp. 1–12.

[32] P. Alikhanifard and N. Tsantalis, "A novel refactoring and semantic aware abstract syntax tree differencing tool and a benchmark for evaluating the accuracy of diff tools," *ACM Transactions on Software Engineering Methodology*, 2024.

[33] M. Martinez, J.-R. Falleri, and M. Monperrus, "Hyperparameter optimization for ast differencing," *IEEE Transactions on Software Engineering*, vol. 49, no. 10, pp. 4814–4828, 2023.

[34] K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang, Y. Liu, and W. Zhao, "CLDiff: Generating concise linked code differences," in *Proceedings of the International Conference on Automated Software Engineering*. ACM, 2018, pp. 679–690.

Authorized licensed use limited to: Universitaet Klagenfurt. Downloaded on March 11,2025 at 15:04:18 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3538326

[35] Y. Yan, N. Cooper, K. Moran, G. Bavota, D. Poshyvanyk, and S. Rich, "Enhancing code understanding for impact analysis by combining transformers and program dependence graphs," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 972–995, 2024.

[36] A. Gyori, S. K. Lahiri, and N. Partush, "Refining interprocedural change-impact analysis using equivalence relations," in *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 318–328.

[37] Q. Hanam, A. Mesbah, and R. Holmes, "Aiding code change understanding with semantic change impact analysis," in *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE, 2019, pp. 202–212.

[38] R. P. Buse and W. R. Weimer, "Automatically documenting program changes," in *Proceedings of the International Conference on Automated Software Engineering*. ACM, 2010, pp. 33–42.

[39] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: How far are we?" in *Proceedings of the International Conference on Automated Software Engineering*. IEEE/ACM, 2018, pp. 373–384.

[40] W. Tao, Y. Zhou, Y. Wang, H. Zhang, H. Wang, and W. Zhang, "Kadel: Knowledge-aware denoising learning for commit message generation," *ACM Trans. on Software Engineering Methodology*, vol. 33, no. 5, 2024.

[41] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2021, pp. 8696–8708.

[42] W. Tao, Y. Wang, E. Shi, L. Du, S. Han, H. Zhang, D. Zhang, and W. Zhang, "On the evaluation of commit message generation models: An experimental study," in *Proceedings of the International Conference on Software Maintenance and Evolution*, 2021, pp. 126–136.

[43] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *Proceedings of the International Conference on Software Engineering*. IEEE, 2023, p. 1430–1442.

[44] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, S. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," in *Proceedings of the International Conference on Learning Representations*. OpenReview.net, 2023.

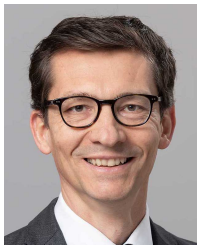
[45] C. S. Xia and L. Zhang, "Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2024, p. 819–831.

[46] D. Hidvégi, K. Etemadi, S. Bobadilla, and M. Monperrus, "Cigar: Cost-efficient program repair with llms," *ArXiv*, vol. abs/2402.06598, 2024.

Beat Fluri is the CTO of Adnovum, a swiss-based software and security engineering company. He earned his PhD in 2008 from the University of Zurich, based on the original research done with ChangeDistiller. In his role as CTO he steers the software engineering lifecyle and technology management of Adnovum.

Michael Würsch is the Head of Integration & IAM Engineering at one of Switzerland's largest health insurance companies. He earned his PhD in 2012 from the University of Zurich, where his research focused on search-driven software engineering, mining software repositories, and Semantic Web technology in software engineering. He is a professional member of ACM.

Martin Pinzger is a professor of software engineering in the Department of Informatics Systems at the University of Klagenfurt, Austria. He is the head of the Software Engineering Research Group and his current research interests are AI for software engineering, mining software repositories, program analysis, and software visualization. He is a member of ACM and a senior member of IEEE.



Harald Gall is a professor of software engineering in the Department of Informatics at the University of Zurich, Switzerland. He held visiting positions at Microsoft Research and the University of Washington in Seattle, USA. His research interests are software evolution, software architecture, software quality, and green software engineering. He is a member of *Academia Europaea*, ACM and IEEE, and a board member of Informatics Europe.

Authorized licensed use limited to: Universitaet Klagenfurt. Downloaded on March 11,2025 at 15:04:18 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.