Precise temporal analysis of source code histories at scale

Quentin Le Dilavrec

COLLEGE MATHS, TELECOMS
DOCTORAL INFORMATIQUE, SIGNAL
BRETAGNE SYSTEMES, ELECTRONIQUE

Université de Rennes

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES

ÉCOLE DOCTORALE N° 601
Mathématiques, Télécommunications, Informatique, Signal, Systèmes, Électronique
Spécialité : *Informatique*

Par
Quentin LE DILAVREC

Precise temporal analysis of source code histories at scale

Analyse précise et à l'échelle des historiques de code source

Thèse présentée et soutenue à Rennes, le 5 février 2024
Unité de recherche : IRISA, Université de Rennes

Rapporteurs avant soutenance :

Martin MONPERRUS Full Professor, KTH, SWEDEN
Martin PINZGER Full Professor, Universität Klagenfurt, AUSTRIA

Composition du Jury :

Examinateurs :	Sandrine BLAZY	Professeure des Universités, Université de Rennes, Rennes, FRANCE
	Jean-Rémy FALLERI	Professeur des Universités, Bordeaux INP, Bordeaux, FRANCE
	Martin MONPERRUS	Full Professor, KTH, Stockholm, SWEDEN
	Martin PINZGER	Full Professor, Universität Klagenfurt, Klagenfurt, AUSTRIA
	Stefano ZACCHIROLI	Professeur des Universités, Télécom Paris, Paris, FRANCE
Dir. de thèse :	Arnaud BLOUIN	Maître de Conférences (HDR), INSA Rennes, Rennes, FRANCE
Co-dir. de thèse :	Jean-Marc JEZEQUEL	Professeur des Universités, Université de Rennes, Rennes, FRANCE
Co-enc. de thèse :	Djamel Eddine KHELLADI	Chargé de Recherche, CNRS, IRISA, Rennes, FRANCE

Remerciements

Je souhaite tout d'abord remercier mes encadrants et co-auteurs, sans qui cette thèse n'aurait su être, pour leur disponibilité, leur travail et leurs conseils stratégiques.
À Djamel Eddine Kheladi, pour sa gentillesse et pour m'avoir soutenu aussi en tant que collègue de bureau, même quand parfois mes idées partaient dans tous les sens.
À Arnaud Blouin, pour son "humour décapant" et ses conseils avisés tant sur l'enseignement que sur les productions scientifiques. À Jean-Marc Jézécquel, pour sa sagesse, ses conseils, et bien sûr le nom HyperAST.
Je suis honoré d'avoir pu travailler avec vous pendant plus de trois ans, et je serai très heureux de travailler à nouveau avec vous.

Je souhaite aussi remercier les professeurs qui m'ont encouragés, tout au long de mes études. Tout particulièrement Gilles Lesventes, sans qui je n'aurais sans doute pas considéré la possibilité d'intégrer le parcours SIF.
Puis Luc Bougé, Martin Quinson, et David Pichardie pour leurs conseils avisés et leur gentillesse durant ce parcours SIF, en particulier, pour m'avoir sensibilisé à l'importance de la mise en avant des contributions scientifiques au-delà des réalisations techniques les supportant.
À Benoit Baudry de m'avoir sensibilisé à l'importance de l'impact des contributions en recherches ; et pour avoir participé à mon CSI pendant 3 ans avec Olivier Barais.

Je souhaite également remercier toute l'équipe DiverSE, qui est pour moi un modèle d'équipe de recherche universitaire, où il fait bon travailler et s'épanouir.
À Oliver Barais, pour avoir été un Super chef d'équipe et avec Benoit Combemale de permettre de maintenir une cohésion, un climat d'entre-aide et d'ouverture.
À Mathieux Acher, pour son enthousiasme, son dévouement à la recherche et pour m'avoir fait découvrir l'équipe.
À Noël Plouzeau, pour son dévouement à l'enseignement et ses formidables histoires qui nous ont fait connaître un peu mieux le monde avant nous.
À Yoann Marquais, pour son humour et son tact.
À Walter Rudematkin, sa convivialité et ses conseils avisés ; "maintenant je comprends mieux pourquoi tu manquais à certains".
À Stéphanie Chalita, pour sa gentillesse.
À Didier Vojtisek, pour les discussions et remarques techniques pleines de sagesse et d'enthousiasme.
À Sophie Maupile, pour sa gentillesse et son assistance dans les démarches administratives.
À June, Gwendal, et Aaron pour avoir organisé les DiverSE coffee.

À Aaron Randriana, pour m'avoir permis de rencontrer un vrai stoïcisme, mais aussi pour avoir partagé ses connaissances littéraires et sa ferveur envers le LISP où au-delà des modes et des idées abstraites, les réalisations concrètes font foi.
À Paul Adam mon premier stagiaire.
Merci à l'équipe des cafés, à l'équipe des goûters, et pour toutes les discussions et moment géniaux qu'on a partagés, Luc Lesoil, Xhevahire Tërnava, Gwendal Jouneaux, June Sallou, Emmanuel Chebbi, Pierre Jeanjean, Paul Temple, Quentin Perez, Théo Giraudet, Romain Lefeuvre, Philémon Houdaille.
Merci également aux nouveaux de l'équipe DiverSE pour leur enthousiasme. Je vous souhaite à toutes et tous d'aboutir à une thèse de doctorat fructueuse.

Pas des moindres, je souhaite remercier ma famille est mes amis qui m'ont toujours soutenu et encouragé, tout particulièrement mon petit frère, Titouan, qui m'a de nombreuses fois servi de cobaye pour travailler sur la pédagogie et l'enseignement.

RÉSUMÉ EN FRANÇAIS

Contexte

Les systèmes de contrôle de versions sont l'une des pierres angulaires du développement des logiciels modernes. En intégrant les modifications entre les versions, ils facilitent la collaboration entre les développeurs et permettent l'intégration et la livraison continues des artefacts logiciels. Au fil du temps, les développeurs produisent de nouvelles versions qui introduisent des fonctionnalités et du code supplémentaires. Par exemple, le noyau Linux compte aujourd'hui plus d'un million de commits, 20 000 fonctionnalités et plus de 15 millions de lignes de code (voir [LM18] et section 11.1.2). Ce qui ralentit le téléchargement, la construction et l'analyse des logiciels [Ach+19]. Par conséquent, il devient essentiel d'améliorer l'efficacité des systèmes qui gèrent les historiques des codes sources des logiciels tout en améliorant la facilité d'utilisation et notre compréhension des historiques déjà existants.

Pour comprendre l'importance et les défis des historiques de code sources, de nombreux parallèles peuvent être établis avec les supports de connaissances antérieurs. Comme les livres, les projets de logiciels contiennent des parties importantes du savoir humain, de la même manière que nous conservons les livres dans des bibliothèques et des archives, nous conservons les historiques des logiciels dans des coffres-forts (*Github Arctic Vault* [Vau20]) et des archives (SoftWare Heritage (SWH) [DZ17]) À l'instar des livres, la croissance de la production et de l'utilisation des logiciels a été rendue possible par l'automatisation. Comme les livres écrits dans des langues anciennes qui sont traduits et adaptés aux langues et aux préoccupations modernes, les logiciels sont modernisés et maintenus; notament par l'usage d'outils qui automatisent des taches de refactoring, de contrôle, et d'ameloriations de la qualité du code. Contrairement aux livres, le développement du code source moderne est massivement collaboratif, des milliers de développeurs modifient des logiciels interdépendants. Contrairement aux livres, les logiciels sont modifiés et publiés à un rythme rapide. Contrairement aux livres, le code source est, en plus d'être lu par des humains, conçu pour être traité et transformé par d'autres programmes, tels que les compilateurs et les systèmes de construction.

Pour que les outils de traitement du code source restent simples, les langages de programmation sont structurés et suivent de nombreuses règles et modèles explicitement définis. Si les règles et les modèles aident les développeurs à définir les programmes avec précision, ils rendent parfois l'adaptation et la modernisation difficiles, de sorte que les développeurs se dotent d'un plus grand nombre d'outils pour automatiser les tâches de

maintenance. Pourtant, en raison des contraintes de ressources et de l'efficacité limitée, les maintenances automatisées existantes sont appliquées à un nombre limité de versions : généralement, la dernière version et quelques têtes de branches sélectionnées. En fait, la maintenance des anciennes versions n'est la plupart du temps assurée que pour les systèmes logiciels critiques, souvent sous la forme de rétro-portages de corrections de bogues de sécurité.

Le développement de logiciels modernes s'appuie sur les *Version Control System (VCS)* pour gérer le code source. Le premier *VCS* remonte à 1972 avec Source Code Control System (SCCS) [Roc75], qui stockait simplement les deltas des fichiers individuels. Depuis 50 ans, de nombreux *VCS* ont été développés, évoluant de manière empirique pour gérer des projets et des pratiques plus exigeants. En 2023, l'*VCS* le plus répandu est *Git*. Il est utilisé pour toutes sortes de projets, et est capable de gérer de grands projets logiciels composés de millions de Lines of Code (LoC), tels que le *noyau Linux*, *Chromium*, ou *LLVM*. Depuis l'avènement de *Git*, les grandes entreprises ont encore amélioré *VCS* pour gérer un grand nombre de projets couvrant l'ensemble de l'entreprise [Met23; PL16]. Enfin, les chercheurs ont conçu des moyens d'archiver efficacement l'historique du code source de millions de projets [DZ17].

Parallèlement à l'avènement de *VCS*, un nouveau domaine de recherche a vu le jour, axé sur l'évolution des logiciels. Les lois de l'évolution des logiciels ont commencé à être énoncées en 1974 par Lehman et Belady, et après des décennies d'affinement, huit lois ont été élaborées [Leh78; Leh79; Leh96; LR03; YM13]. En 1976, l'auteur [Swa76] a classé la maintenance des logiciels en fonction du contexte et de la cause de l'activité de maintenance. En 1994, Parnas présente le vieillissement des logiciels [Par94], établissant un parallèle entre le vieillissement des logiciels et celui des organismes vivants.

Avec tous ces outils et une compréhension approfondie de l'évolution des logiciels, il est désormais beaucoup plus facile de créer, de contribuer et de partager des systèmes logiciels. Les forges logicielles, telles que *GitHub* ou *Gitlab*, constituent l'une des dernières améliorations sur le front de l'accessibilité, avec des fonctionnalités et des outils complets qui aident à organiser et à automatiser le développement. Si l'on ajoute les avantages du développement open-source, [Hec99], il a conduit à une forte augmentation des projets open-source. Par exemple, les références SWH s'élèvent à environ 200 millions d'origines *Git* (y compris les forks), dont 188 millions sur *GitHub* (*cf.* section 11.2).

Problème

Fournir en permanence des produits et des services robustes sur des systèmes logiciels massifs et complexes nécessite des analyses et des tests approfondis. Par conséquent, l'exécution de toutes les analyses et de tous les tests à chaque changement constitue un défi, voire même un impossibilité. Les systèmes de construction et les outils d'analyse

existants s'attaquent déjà à cette complexité par des approches incrémentales, soit en modifiant de manière incrémentale un état transitoire, soit en mettant en cache des artefacts intermédiaires. Cependant, les approches existantes considèrent des artefacts à gros grain (le plus souvent au niveau du fichier) et une représentation basée sur le texte (avec peu d'accès à la sémantique). Dans cette thèse, nous démontrons qu'il est possible d'améliorer de manière significative les approches incrémentales en considérant des éléments à grain plus fin tout en adaptant les analyses existantes.

Cette thèse vise donc à relever divers défis scientifiques concernant l'extensibilité des approches et des analyses logicielles à de grands projets logiciels industriels. Le premier défi que j'aborde concerne l'analyse temporelle de l'historique des codes sources par le biais de coévolutions fines, afin d'aider à la compréhension et à l'automatisation des codes. Ces coévolutions constituent un défi en raison des relations complexes entre les éléments du code et de la quantité de bruit qui les entoure. L'identification des coévolutions qui s'étendent sur plusieurs versions et commits est particulièrement coûteuse, augmentant de manière exponentielle avec la taille de la fenêtre de l'historique du code. Dans les coévolutions, les dimensions spatiales et temporelles de l'analyse du code source s'additionnent, ce qui rend son identification plus difficile à mettre à l'échelle sur de longues périodes de l'histoire du logiciel. Ce premier défi a permis de réaliser que les représentations structurées actuelles du code source ne permettent pas d'effectuer des analyses temporelles efficaces. En effet, le deuxième défi concerne l'inefficacité des analyses temporelles actuelles du code source, qui ne parviennent pas à exploiter la redondance temporelle et spatiale du code source dans l'historique des logiciels. Dans la plupart des historiques de logiciels, la quantité de changements apportés par chaque commit est faible par rapport à la taille de l'ensemble de la base de code, c'est-à-dire quelques lignes de code parmi des millions.

Contributions

Concrètement, il nous manque une représentation du code source qui permette de partager des morceaux de code inchangés entre plusieurs versions. Ainsi, la mémoire est épargnée et le partage des résultats intermédiaires est possible, ce qui rend les composantes des analyses temporelles incrémentielles. Les contributions majeures de cette thèse comprennent la fourniture d'une nouvelle structure de données qui bénéficie à l'analyse des deux dimensions, à savoir spatiale et temporelle. Pour la dimension spatiale, je l'ai d'abord démontrée en relevant le défi de trouver toutes les références à des déclarations données. Plutôt que de créer et de maintenir une table d'association pour chaque version, je propose de m'appuyer sur des oracles probabilistes locaux pour naviguer efficacement dans le code en élaguant les branches infructueuses de la recherche. En ce qui concerne la dimension temporelle, j'en fais la démonstration en relevant le défi de la différenciation des versions du code source. Plutôt que de considérer des arbres de code individuels (qui doivent

également être analysés individuellement), je propose d'exploiter les éléments de code partagés pour lazifier les algorithmes de différenciation, ce qui permet de les adapter à des milliers de commits composés de millions de nœuds.

Par conséquent, tout au long de cette thèse, le thème le plus récurrent est l'importance de la mise à l'échelle d'analyse de grands projets "industriels" de code source.

Chapters 2 to 4 présente le contexte et l'état de l'art concernant la gestion et l'analyse des historiques de code source, à la fois à partir de la littérature académique et des outils de l'industrie. Chapter 2 couvre la gestion et la formalisation des évolutions des logiciels. Chapter 3 présente des approches qui analysent le code source à un moment donné (en dehors de toute considération temporelle), telles que l'analyse statique et dynamique, notamment la résolution de noms. Enfin, chapter 4 présente des analyses temporelles de l'histoire des codes sources, en combinant les aspects développés dans chapters 2 and 3.

Je présente ensuite trois contributions, pour mieux entretenir l'historique des codes sources. La première contribution (chapter 5) sert à montrer qu'il est effectivement possible d'analyser les changements de code et leurs impacts à un niveau fin, en déduisant les relations de causalité structurelles et fonctionnelles entre les changements. Ensuite, chapter 6 présente comment l'analyse des historiques de code peut être rendue efficace en représentant le code source dans une structure (nommée *HyperAST*) qui permet d'exploiter la redondance dans l'espace et le temps tout en rendant les analyses notables incrémentales. Sur la dimension spatiale, chapter 7 présente une approche qui calcule de manière incrémentale les impacts possibles de ces changements. Sur la dimension temporelle, chapter 8 présente une approche (nommée *HyperDiff*) qui calcule de manière incrémentale les changements de l'arbre du code source à un niveau fin.

Chapter 9 discute des considérations techniques importantes des contributions présentées, en particulier en ce qui concerne la mise en œuvre de *HyperAST*, de *HyperDiff* et de leurs composants. Chapter 9 présente également des cas d'utilisation de l'HyperAST : le suivi du code source à travers un long historique, le calcul efficace des métriques de lots avec des requêtes dynamiques, et la notation des devoirs de programmation. Chapter 10 conclut la thèse et présente quelques perspectives.

Table of Contents

Chapter 1

Introduction

> "Calculating Machines comprise various pieces of mechanism for assisting the human mind in executing the operations of arithmetic. Some few of these perform the whole operation without any mental attention when once the given numbers have been put into the machine."
>
> — Charles Babbage, *Passages from the Life of a Philosopher (1864), p. 42*

VCS are one of the corner stones of modern software development. By tacking changes across versions, they facilitate developers collaboration, and enable continuous integration and delivery of software artifacts. Over time, developers produce new versions that introduce additional features and code. For example, the Linux kernel now has more than 1M commits, 20K features, and more than 15M lines of code (see [LM18] and section 11.1.2). Making it slow to download,[1] build and analyze [Ach+19]. Therefore, it becomes essential to improve the efficiency of systems that manage software source code histories while also improving usability and our understanding of already existing histories.

To understand the importance and challenges of source code histories, many parallels can be drawn with previous knowledge supports. Like books, software projects hold important parts of human knowledge. In the same way we keep book in libraries and archives, we keep software histories in vaults (*Github Arctic Vault* [Vau20]) and archives (SWH [DZ17]). Like books, the growth of software production and usage was enabled by automation. Like books written in ancient languages that are translated and adapted to modern languages and concerns, software are also modernized and maintained through time to stay relevant. Unlike books, modern source code development is massively collaborative, thousands of developers interact with dependent software. Unlike books, software are modified and released at a fast pace. Unlike books, source code is in addition to being read by humans, designed to be processed and transformed by other programs, such as compilers and build systems.

To keep source code processing tools simple, programming languages are structured and follow numerous explicitly defined rules and patterns. While rules and patterns help developers define programs precisely, they sometimes make adaptation and modernization cumbersome, so developers help themselves with even more tool to automate maintenance tasks. Yet, due to resources constraints and limited efficiency, existing automated maintenances are applied on a limited number of versions: usually, the latest version and some

[1] https://github.blog/2018-03-05-measuring-the-many-sizes-of-a-git-repository/

selected branch heads. Actually, maintenance of older versions are most of the time only provided for critical software systems, often in the form of back ports of security bug fixes.

Modern software development relies on *VCS* to manage source code. The first *VCS* can be traced back to 1972 with SCCS [Roc75], that simply stored delta of individual files. Since 50 years, many *VCS* have been developed, evolving empirically to manage more demanding projects and practices. As of 2023 the most widespread *VCS* is *Git*. It is used for all kinds of projects, and is able to handle large software projects made of millions LoC, such as the *Linux kernel*, *Chromium*, or *LLVM*. Since the advent of *Git*, large companies have further improved *VCS* to manage large number of projects spanning over their entire company [Met23; PL16]. Finally, researchers have devised ways of efficiently archiving the source code histories of millions of projects [DZ17].

In parallel to the advent of *VCS*, came a new field of research focusing on the evolution of software. Laws of software evolutions started being enunciated in 1974 by Lehman and Belady, and following decades of refinement, eight laws were developed. [Leh78; Leh79; Leh96; LR03; YM13]. In 1976, Swanson categorized maintenance of software depending on the context and cause of the maintenance activity. In 1994, Parnas introduces software aging [Par94], tracing a parallel between aging in software and living organisms.

With all these tools and a thorough understanding of software evolution, it is now much easier to create, contribute and share software systems. One of the latest improvement on the front of accessibility are software forges, such as *GitHub* or *Gitlab*, with extensive features and tools helping with organizing and automating development. Adding up with advantages of open-source development [Hec99], it has led to a large increase of open-source projects. For example, SWH references is around 200 millions *Git* origins (including forks) where 188 million of them are on *GitHub* (*cf.* section 11.2).

Continuously delivering robust products and services on massive and complex software systems requires extensive analyses and tests. Therefore, executing every analyses and tests at every change is challenging or downright impractical. Existing build systems and analysis tools already tackle this complexity through incremental approaches, either incrementally mutating a transient state or caching intermediate artifacts. Yet, existing approaches still consider coarse grained artifacts (often at file level) and text based representation (little access to semantic). In this thesis, we calm that it is possible to significantly improve incremental approaches by considering finer grained elements while adapting existing analyses.

This thesis aims to tackle various scientific challenges around the scalability of software approaches and analyzes to large industrial software projects. The initial challenge I address concerns the temporal analysis of source code histories through fine-grained co-evolutions, to help with code understanding and automation. These co-evolutions are challenging due to the intricate relationships between code elements and the amount of

noise that surrounds them.[2] Identifying co-evolutions that spans multiple versions and commits is particularly expensive, exponentially growing with the window's size of the code history. In co-evolutions, both the spatial and temporal dimensions of source code analysis compound, making its identification harder to scale over long software histories. This initial challenge led to the realization that current structured source code representations are not enabling efficient temporal analyzes. Indeed, the second challenge concerns the inefficiency of current temporal source code analyzes, which fail to exploit temporal and spatial source code redundancy in software histories. In most software histories, the quantity of changes each commit brings is small compared to the size of the whole code base, *e.g.,* a single line in millions of lines of code. Concretely, we are missing a source code representation that supports sharing unchanged pieces of code over multiple versions. Thus, sparing memory and enabling the sharing of intermediate results, consequently, making components of temporal analyses incremental. The major contributions provided in this thesis includes providing a new data structure that at least benefits analysis of both dimensions, namely spatial and temporal. For the spatial dimension, I first demonstrate it by addressing the challenge of finding all references to given declarations. Rather that creating and maintaining an association table for each version, I propose to rely on local probabilistic oracles to efficiently navigate code *i.e.,* pruning fruitless branches from the search. For the temporal dimension, I then demonstrate it by addressing the challenge of diffing source code versions. Rather than considering individual tree of code (that must be also individually parsed), I propose to leverage the shared code elements to lazify the diffing algorithms, making it scale to thousands of commits made of millions of nodes.

Consequently, throughout this thesis, the most recurring theme revolves around the importance of scaling to large "industrial" source code projects.

Chapters 2 to 4 present the background and state-of-the-art regarding the management and analysis of source code histories, both from the academic literature and the industry tools. Chapter 2 covers the management and formalization of software evolutions. Chapter 3 presents approaches that analyze source code at a given moment (outside temporal considerations), such as static and dynamic analysis, notably name resolution. Finally, chapter 4 presents temporal analyses of source code histories, combining the aspects developed in chapters 2 and 3.

I then present three contributions, toward the goal of better maintaining source code histories. The first contribution (chapter 5) serves to show that it is indeed possible to analyze code changes and their fine-grained impacts, deducing structural and functional causality relations between changes. Then, chapter 6 presents how analyzing code histories can be made efficient by representing source code in a structure (named *HyperAST*) that allows to exploit redundancy in space and time while making notable analyses incremental.

[2] It seems pretty close to superposition in deep learning systems: `https://transformer-circuits.pub/2022/toy_model/index.html`

On the spatial dimension, chapter 7 presents an approach that incrementally computes the possible impacts of these changes. On the temporal dimension, chapter 8 presents an approach (named *HyperDiff*) that incrementally computes fine-grained source code tree changes.

Chapter 9 discusses important technical considerations of presented contributions, particularly regarding the implementation of the *HyperAST*, the *HyperDiff* and their components. Chapter 9 also presents use cases for the HyperAST: at tracking source code throughout a long history, efficiently computing batch metrics with dynamic queries, and grading programming homework. Chapter 10 concludes the thesis and presents some perspectives.

Part I

Background & State-Of-The-Art

---

(1) We cannot assume that the old stuff is known and didn't work. If it didn't work, we have to find out why. Often it is because it wasn't tried.

(2) We cannot assume that the old stuff will work. Sometimes widely held beliefs are wrong.

(3) We cannot ignore the splinter software engineering groups. Together they outnumber the people who will read our papers or come to our conferences.

(4) Model products are a must. If we cannot illustrate a principle with a real product, there may well be something wrong with the principle, Even if the principle is right, without real models, the technology won't transfer. Practitioners imitate what they see in other products. If we want our ideas to catch on, we have to put them into products. There is a legitimate, honorable and important place for researchers who don't invent new ideas but, instead, apply, demonstrate, and evaluate old ones.

(5) We need to make the phrase "software engineer" mean something. Until we have professional standards, reasonably standardised educational requirements, and a professional identity, we have no right to use the phrase, "Software Engineering".

– David Lorge Parnas, *Software Aging* (1994)
sec. 10. Conclusions for our profession

Chapter 2

Software evolution

This chapter presents the state-of-the-art and concepts regarding the management of source code histories and study of software evolutions. Section 2.1 covers how source code histories are managed and processed by recounting the emergence VCS (section 2.1.1), then detailing the one that now prevails, namely *Git*. Section 2.2 is devoted to the definition of software evolution and surrounding concepts.

2.1 Source Code Management systems

Many system management tools exist. Configuration managers, such as *Ansible* or *Chef* are used to configure servers and pools of computers. Package managers either at the system level like *apt* or *pacman*, or at the application level like *maven* or *npm* help with retrieving binary and source code artifacts stored in registries. In this thesis, I will focus on Source Code Management (SCM) systems, also synonym to Version Control Systems (VCS). To some extent I will also consider the integration into software forges and CI/CD systems.

VCS provide developers with both the development history of their software systems and control over the evolutions of their software systems by keeping track of their changes and by providing multiple features to leverage development histories. Examples of widely-used VCS are *Git*,[1] *Mercurial*,[2] and *SVN*.[3] Taking *Git* as example, developers do commits, *i.e.,* set of changes in their source code associated with a description message. The set of commits composes the history of the versioned software. Figure 2.1b gives an excerpt of an open-source project versioned using *Git* and visualized with *Github*. Each commit has an author, a description message, a date, references to issue tracking system (*e.g.* `#3394`), and when part of a forge can incorporate information related to tools that automatically builds, checks elements of the software system on each commit (here the ✓ and × symbols that point to such tools). Modern VCS enable many fundamental process and practices, that facilitate collaborative software development by allowing each developer to work concurrently on different source states that can be merged once in a satisfying state. For example to develop the Linux kernel, a decentralized approach is preferred. Here, progress

[1] `https://git-scm.com`
[2] `https://www.mercurial-scm.org`
[3] `https://subversion.apache.org`

is shared by pulling commits from each other repository there is no central source of truth.[a] On the flip-side, at Google or Meta, a trunk (a branch in a specific repository instance) is designated while others source states are short-lived working branches. Short-lived branches can be merged into the trunk and deployed after having passed automated tests and reviews of collaborators, *i.e.,* Continuous Integration (CI)/Continuous Deployment (CD).

VCS are also an essential part of software forges. A change made to source stored in the forge (after a push) triggers packaging and distribution once a set of automated checks succeed and changes get positively reviewed (Pull request, CI). Each new version is qualified with a version number that document the nature of the change (semantic versioning), in addition to a summary of notable changes in a human-readable format (changelog) that can be automatically –at least in part– generated using metadata from commits since the last release. The source code history and neighboring artifacts can be navigated with search tools. A bug report or a contribution (Issue, Pull request) can mention specific code ranges, then these code ranges can be observed in context (in their file, with a diff, or a blame), in conjunction commit messages can reference existing issues (marking as fixed). In addition, these issues and pull requests can be automatically submitted by bots, to notably upgrade dependencies or notify uses of vulnerable code.

(a) Git command

(b) Github web app

Figure 2.1 – Commit summaries part of the git history of a software system

2.1.1 Emergence of version control systems

Version control is an old concern. Looking at the historical developments of VCS, we can observe the empirical adoption of new concepts and approaches to version source code, notably through their open-source counterpart (in bold).

It started with local VCS, versioning individual files as deltas, where collaboration took the form of acquiring locks on files. *Source Code Control System (SCCS)* [Roc75] started

being developed in 1972 for a first public release in 1977, using interleaved deltas. Then came ***Revision Control System (RCS)*** [Tic85], released in 1982, that used reverse deltas to improve performances while on the latest versions [Tic62].

Then came the centralized client-server SCM systems. ***Concurrent Versioning System (CVS)***, first released in 1990, which allows multiple *CVS* clients to synchronize with a single *CVS* server which internally uses *RCS*. ***Subversion (SVN)*** first released in 2000, with significant improvements compared to *CVS*, most notably allowing to commit multiple changed files in a single transaction.

Finally, came the distributed SCM systems, with *BitKeeper* in 2000 followed by ***Git*** and ***Mercurial*** [4] in 2005, due mainly to license issues. While *BitKeeper*, *Mercurial*, and previous VCS are based on the changesets model, *Git* is on contrary designed like backup snapshots of a file system (a snapshot per commit).

Name	Creation	Repr.	State	Granularity
CVS	1990	Changeset	Centralized	file
SVN	2000	Changeset	Centralized	tree
BitKeeper	2002	Changeset	Distributed	tree
Mercurial	2005	Changeset	Distributed	tree
Git	2005	Snapshot	Distributed	tree
Piper	2006	Snapshot	Centralized	tree
Sapling	2006	Snapshot	Hybrid	tree
SWH	2016	Snapshot	Distributed	tree

Table 2.1 – Comparison of a few notable Version Control Systems.[5]

Nowadays, two kinds of versioning systems are mainly distinguished:
ChangeSet where only changes are stored, *i.e.,* set of actions on source code, and
Snapshot where only the state of the source code is stored, *i.e.,* content of files and structure of directories.
This distinction is crucial to understand capabilities of modern *VCS* , yet it is often misunderstood. Therefore, in the next section, I am going to explain how *Snapshot VCS* work. Besides, there is a notable *Github* blog post [6] clarifying *Snapshot* nature of *Git*.

[4] `https://www.mercurial-scm.org`

[5] You can also find a well maintained comparison in `https://en.wikipedia.org/wiki/Comparison_of_version-control_software`.

[6] `https://github.blog/2020-12-17-commits-are-snapshots-not-diffs/`

It is also possible to distinguish source code repositories into mono- and micro-repositories. Very large repositories (Millions of lines of code) spanning over entire organizations are often considered as mono-repositories, but other criteria can be used. Mono-repositories are reportedly used in large organizations, such as *Google* or *Meta/Facebook* [PL16; Met23]. The *Linux kernel* can also be considered as a mono-repo, but is more like a mono-tree [a] where there is no central repository. Notable multi-repositories are ones of *Apache* or *Oracle*, and reportedly at *Amazon* or *Netflix* [Bro19].

[a] `https://blog.ffwll.ch/2017/08/github-why-cant-host-the-kernel.html`

2.1.2 Snapshot VCS

Git is currently the most widespread (*Snapshot*) VCS,[7] so I will to focus on *Git* and present its most notable principles, core data structure, and vocabulary. I will also briefly mention improvements made in clones/forks of *Git*. But let me first introduce the Merkle DAG structure [Mer87], as it is a central component of *Git* that enables the robust and efficient storage of snapshots.

Merkle DAG

A Merkle DAG is a data structure similar to a Merkle Tree [Mer87], but with nodes holding a payload and no balancing requirements. Each node is uniquely identified by a cryptographic hash of its content (the hash function must be shared), and child nodes can easily be referenced in the payload of nodes, along with other data. Consequently, structurally identical nodes are deduplicated, effectively making it a Direct Acyclic Graph (DAG). This identification scheme is especially useful for archival [DZ17], and is categorized by Di Cosmo and Zacchiroli as *intrinsic* identifiers (in opposition to *extrinsic* identifiers that necessitate a registry).

However, a Merkle DAG has limitations regarding its compression efficiency: 1) the diversity of stored content, *i.e.,* only structurally identical nodes are shared, 2) the size of payloads of nodes relative to the size of identifiers, *i.e.,* the memory overhead of the unique identifying hashs must be compensated by the sharing of identical content; To make collisions as improbable as possible, the unique identifying hashs must be large, tenth of bytes long and scaled in proportion to the expected number of unique nodes. Additional metadata and contextual intermediary results must be stored aside (akin to columnar databases) due to nodes being shared and append-only. The internal fork of *Git* by *Github* reportedly uses additional technics to compress git objects during packing [Bla22].

[7] *Git* represents 97.5% of origins reported SWH, section 11.2 contains scraping results

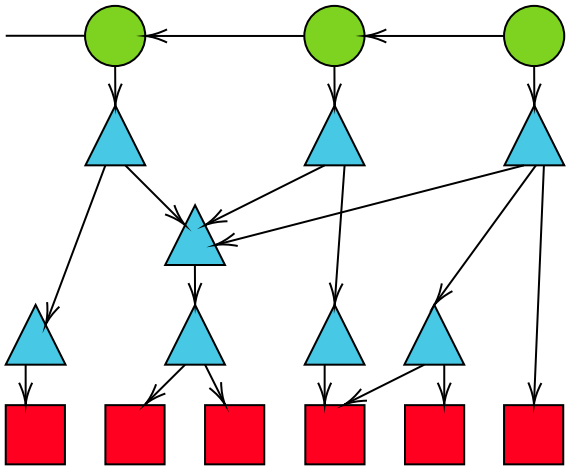


Figure 2.2 – Example of a Git Merkle DAG.
Legend: circle: Commit, triangle: Tree, square: Blob.

The Git VCS

In Git, elements of the MerkleDAG are named *Objects*.[8] *Objects* have a unique identifier called *Oid* that is the hash of the content of the *Object*. Figure 2.2 depicts the three major kinds of *Objects*: a *Blob* object (square) corresponds to the content of a file; a *Tree* object (triangle) corresponds to a directory, it maps by name other *Tree* objects but also *Blob* objects; a *Commit* object (circle) is a snapshot of the codebase. It points to a *Tree* and its parent *Commit* objects. For the sake of simplicity, these objects are referenced as *Blob*, *Tree*, and *Commit* in the rest of the thesis.

The Git model is simple and allows some robust and flexible merge heuristics, as explained in the FAQ of Git [9]:

> "Relying on explicit rename tracking makes it impossible to merge two trees that have done exactly the same thing, except one did it as a patch (create/delete) and one did it using some other heuristics"

and by Linus Torvalds himself [10]:

> "That's because GIT fundamentally doesn't do the 'delta-within-a-file' model."

Git focuses on storing content. Changes are an optimization and search concern. In cases where there is a particular intent behind a change, it should be written in the commit message, technics exist to formalize such intents, such as conventional commits. Separating the intent from modifications allows to search code while reinterpreting the history with new approaches, *e.g.,* different diff algorithms as we will see in the next section. On the performance side, change related technics are used to compress the history, for example, blobs are compressed by default, and existing objects can be packed in a compressed

[8] https://git-scm.com/docs/gitglossary

[9] `https://archive.kernel.org/oldwiki/git.wiki.kernel.org/index.php/GitFaq.html#Why_does_Git_not_.22track.22_renames.3F`

[10] `https://marc.info/?l=linux-kernel&m=111314792424707`

format.

Moreover, to accommodate additional constraints, custom implementations of *Git* emerged with significant modifications, particularly on the infrastructure side:

— *Google Piper [PL16]*, the internal VCS of *Google* hosting their famous mono-repo, and reportedly offering access control and partial fetch [PL16],

— *Meta Sapling*, the recently open-sourced VCS of Meta, also hosting their own mono-repo,

— *Software heritage (SWH) [DZ17]* from *INRIA*, which focuses on archiving source code,

— *GitHub*'s internal fork of *Git*, they contribute to upstream *Git* (partially due to licensing).

Software Heritage [DZ17] uses different naming conventions: Blob → file content, Tree → directory, Commit → revision, Tag → release.

2.1.3 Working with source code changes: diff, merge, blame

VCS provide developers with merging and diffing tools to help developers manage and monitor changes. As explained in the previous section Snapshot SCM systems (such as *Git*), do not store individual changes but compute them on demand, *i.e.,* changes can be reinterpreted depending on context. For example with *Git*, the command `git diff` has four preconfigured algorithms [NHM20]: myers (default)[Mye86], minimal, patience, or histogram; and can use external tools by setting an environment variable with an executable.[11] Approaches computing diffs can be classified by their structuration degree, *i.e.,* the amount of semantic information used to structure and compare diffed content.

Figure 2.3b illustrates a rendered diff output (*patch* [MES02]) where added lines are green and start with the '+' symbol, while removed lines are red and start with '-'.

The most general and widespread diff approaches process source code like any other text (*i.e.,* a list of characters), with commits considered as sets of text files associated to paths.

Asaduzzaman *et al.* [Asa+13], Canfora *et al.* [CCD08] and Reiss *et al.* [Rei08] proposed a language-independent techniques for diffing. But sometimes, considering code as text is not enough to consistently obtain the high quality diffs that fits the original intent of the developer and refer to language specific constructs.

Tree Diff

Apart from diffs on text, it is also possible to process the structured tree representation of source code into a diff, also called an edit script. Zhang and Shasha [ZS89] defined an

[11] `https://git-scm.com/docs/git-diff#Documentation/git-diff.txt-diffexternal`

(a) Git command (b) Github web app

Figure 2.3 – A commit that shows added and removed text chunks using a syntactic diff algorithm

edit distance between pairs of trees, using insertion, deletions and updates. They also proposed an algorithm to compute this distance optimally, it is named the ZS algorithm in the literature. Chawathe *et al.* proposed to compute edit scripts on trees and defined complex actions that can operate on subtrees, such as move [Cha+96] (in the literature it is called the Chawathe algorithm), copy and glue [CG97], thus, shortening the resulting diff. Pawlik *et al.* [PA11] proposed to improve the efficiency of worst case executions from the ZS algorithm. Duley *et al.* [DSK12] presented a diff for Verilog HDL files. Hashimoto *et al.* [HM08] proposed to further preprocess ASTs: pruning common subtrees, and collapsing certain syntactic categories into hash digests, then using the ZS algorithm [ZS89]. Nguyen *et al.* [Ngu+11] also proposed to compute a diff, but they focused more on finding cloning actions rather than a complete diff.

Fluri *et al.* [Flu+07] proposed *ChangeDistiller*, a tool and algorithm improving over [Cha+96] for computing diffs. Similarly, Falleri *et al.* [Fal+14] proposed a tool and algorithm, called *GumTree*, that extends the matching phase [Cha+96] by first mapping subtrees greedily (inspired by [CAM02]), then falling back to the ZS algorithm, thus reducing the number of comparisons and the average complexity. Higo *et al.* [HOK17] proposed to integrate the action of copy-and-paste to make the diff easier to understand for developers. Matsumoto *et al.* [MHK19] also proposed an extension of GumTree by incorporating information of line differences in addition to the AST to also ease

comprehension. Other approaches aim at improving the diffs that GumTree produces with additional steps [DP16; Fri+18; Hua+18].

All these existing approaches focus on efficiently computing a diff between two files. Yet, when developers ask for a `git diff` they do so on entire commits picked from a source code history (Git). Since a Git commit can contain numerous files, diffing entire commits directly faces scalability issues.

Tsantalis *et al.* [TKD20] proposed an approach for detecting refactorings, called *RefactoringMiner*. Refactorings are high level changes that do not by themselves fully explain changes to a codebase. Moreover, *RefactoringMiner* mostly focuses on *Java*, while some other recent extensions make it work on *Python* [Atw+21] and *Kotlin* [Zar20].

Similarly, for multiple languages, Silva *et al.* [Sil+20] with *RefDiff* also proposed an approach capable of finding refactorings.

Finally, Falleri *et al.* [FMM15] combined *Spoon* [Paw+15] with *GumTree* [Fal+14], allowing them to handle entire *Java* projects *i.e.,*, diffing multiple files.

(Structured) Merge

In addition to Diff, an essential part of modern collaborative development is the ability to merge contributions automatically or at least semi-automatically, *i.e.,* only acting on merge conflicts. Actually, merging can be considered as an extension of Diffing [MES02; Men02].Textual merges, working on lines of text are widely adopted [MES02; Men02]. For example, *Git* provides three merge algorithms all configurable with diff-algorithms: three-way [KKP07], recursive, and octopus. For AST merges there is notably an approach presented by Larsén et al. [Lar+22] that leverage Gumtree to reduce false collisions as they append in line based merges.

Real-time collaboration is also a merge problem. Without getting into details (of the CAP theorem), in a distributed context *i.e.,* each user has a version of the data, when users share their modifications conflicts might arise, that would need an intervention and would hinder the editing process. So merge conflicts must be "resolved" automatically, more specifically, it refers to Conflict-free Replicated Data Types (CRDT) [Sha+11; KB17].

Blame / Code Tracking

Git also provides a blame subcommand (`git blame`) which enables to track textual changes throughout consecutive commits. It provides options to track duplicates and match specific textual patterns.

Synonymously to blame, we can track changes. Tracking is often used for more structured approaches [HHK20; HMK11; GAS19] that track code elements (code blocks, attributes, methods, ...) in a source code history.

IntelliJ provides a code tracking feature capable of following renames. [HMK11]

improves capabilities of *Git* with tracking methods by isolating said method in individual files, it has limited capabilities in cases of short methods. *Cregit* [GAS19] further improves diff and blames from *Git* by first preprocessing the source code (C/C++), *i.e.,* it parses the source code text and feed the usual diff algorithm with tokens (one per line) corresponding to code elements along with their type in the syntax tree. Similarly, *FinerGit* [HHK20] preprocess source code to improve the textual diff, putting a token per line and annotating them with their type. Contrary to *Cregit*, it focuses on *Java* source code.

To better handle complex refactorings (not just renames), additional approaches where devised: *Code Shovel* [Gru+21] is a multistaged approach that allows to track a given method up to its introduction, it does not use tree diffs nor advanced semantic information but focuses on specific method change heuristics. *Code Shovel* seems to provide better results and performances than *FinerGit* to track methods. Finally, while it seem to be able to easily scale out, it might not be efficient to track hundreds of methods on large code bases. *Code Tracker* [JT22] leverages *RefactoringMiner* to track methods and classes. It outperforms *Code Shovel*. *SEAL* [Sat+23] improves blame quality using dataflow analysis. It is a followup to *Cregit*.

Providing advanced maintenance features on source code histories still remains a challenge for large complex projects. Indeed, there are no approaches that that scale, while also representing fine-grained code together with full code analysis.

2.2 Source code evolution

The goal of a software evolution is to adapt the different artifacts of the software to the ever-changing user requirements and environment. An evolution consists of one or multiple changes that developers apply on the software artifacts. For example, in Figure 2.3b, a change in the imports was committed to the class.

Software evolution can occur under different forms to achieve various purposes. Various types of evolution have been categorized in the past [Swa76; LS80], under the name of maintenance evolutions:

— *Corrective* evolution refers to changes that correct any discovered problems, errors, failures, and inconsistencies, etc.

— *Adaptive* evolution aims to keep a software product usable in a changing environment.

— *Perfective* evolution aims to improve functionalities, performance, reliability, or to increase the maintainability of a software.

In practice, developers can set up guidelines to rationalize contributions, and specify a concrete format for commits messages.[12] The change type [13] is often the first information,

[12] `https://github.com/angular/angular/blob/22b96b9/CONTRIBUTING.md#commit-message-format`

[13] `https://github.com/angular/angular/blob/22b96b9/CONTRIBUTING.md#type`

optionally followed by a scope, a subject, and a detailed message. Some widely used change types are `feat` when adding a feature, `fix` when fixing a bug, `refactor` when making a structural change that should not change behavior. The change type is sometime mixed with the scope, *e.g., docs* that corresponds to documentation only changes as it sometimes makes little sense to specify the precise nature of the change. Formatting commits messages has also the advantage of enabling further processing by other tool, for example, to automatically generate *changelogs*, *i.e.,* a human-readable list of changes with categories directly corresponding to change types. It can also help with semantic versioning [Pre13; RVV14] aka `MAJOR.MINOR.PATCH` by automatically checking or incrementing the version. Indeed, a fix corresponds to a `PATCH`, while a new feature corresponds to a `MINOR`, and it is also possible to notify breaking changes, *i.e.,* `MAJOR`.

Table 2.2 presents a comparison between change types form conventional commits and the maintenance evolution classification.

Change type	Corrective	Adaptive	Perfective
Bug Fix	✓		
Refactoring		✓	
Optimization			✓

Table 2.2 – Classification of types of maintenance evolutions

Change types can also be classified depending on the nature of its impacts, as shown in table 2.3. It can either be:

— functional, *e.g.,* break or fix a test,

— operational, *e.g.,* improve performances, or

— developmental, *e.g.,* rename a class.

Change type	Functional	Operational	Developmental
Feature Addition	+	±	±
Bug Fix	+	±	±
Refactoring	=	±	+
Optimization	=	+	±

Table 2.3 – Nature of changes impacts

One popular type of changes is *refactoring*. Software refactoring is the object-oriented term of restructuring as it was introduced by Opdyke [Opd92]. Refactoring is any modification to the software with the two purposes: 1) to make it easier to understand and to change or 2) to make it less susceptible to errors when future changes are introduced

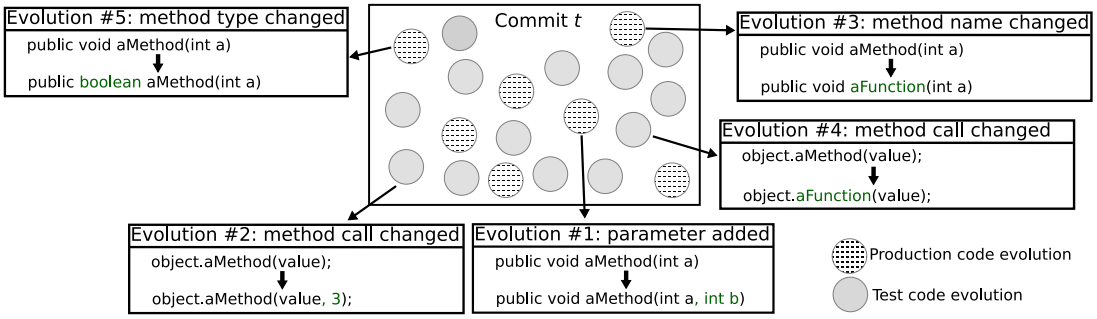


Figure 2.4 – Example of evolutions in a commit identified by evolutions detection tools, such as RefactoringMiner [Tsa+18a; TKD20].

[Arn86; Opd92]. Overall, a refactoring improves quality and facilitates future adaptations and extensions, but without having a functional effect, or in the worst case without introducing any new behavior on the conceptual level [MT04; Sun+01].

Regardless of the type or intent of changes, ultimately, a change is an evolution to the code varying in size from fine to coarse grained, That we can define as follow:

Definition 1 – Source code evolution. We refer to changes in source code as evolutions spotted from code histories, for example, between commits or releases. The literature classifies evolutions into two groups that correspond to two levels of granularity:

— Atomic evolutions in the form of insertions, deletions, or updates of an AST element [Fal+14];

— Complex evolutions take the form of composition of atomic evolutions. For example, deleting a piece of code and inserting the same piece somewhere else can be interpreted as a move. Another example of complex evolutions are refactoring operations [Fow99], such as *extract class/method* or *push/pull method*.

2.3 Conclusion

This first background chapter presented the "temporal" dimension of source code analysis, and presented VCS, while detailing the most prevalent, namely *Git*. Then it presented major uses of VCS targeted toward studying and understanding source code change and evolution: Diffing, Merging and Tracking. Finally, it presented the literature on source code evolution and defined them.

Next chapter will present the "spatial" dimension of source code analysis, and presented approaches that allow to relate code elements at a given commit.

Chapter 3

Processing and Analyzing Software

Software systems are often large and complex. Their source code easily reach millions of lines, written by many contributors, spanning over numerous concerns. Build systems process those millions of lines into usable artifacts, targeting diverse platforms and hardware. These three factors (size, complexity, and diversity), make it challenging to predict the behavior of all or part of a software system (or change to it). To help themselves, software engineers use additional pieces of software to analyze and test source code in different environments. Nonetheless, analyzing and testing come at a cost, mainly time but also computing resources.

Assessing source code quality is an important part of producing complex yet reliable software systems. It can be achieved through various means, often categorized between static and dynamic analyzes or as hybrids of both. It is also possible to consider the dichotomy between simulation and emulation, respectively checking abstract properties, *e.g.,* termination and computing concrete values in an emulated context, *e.g.,* inputs. Static and dynamic analyzes often complement each other, due to trade-offs among precision, recall, and compute/development time. Concretely, static analysis is best suited to prove simple properties for any inputs, while dynamic analysis takes a concrete input to compute a concrete result.

In this context, performance of analyzes is particularly challenging as it often limits the interactivity of analyzes. We will often be able to classify three possibilities to make an analysis/process scale: doing less, adding compute power, trading compute against memory. In many situations, build and analysis tasks can have (or can be made into) independent components. It will be mentioned at multiple occasions in this chapter, especially for the Hybrid analyzes.

Section 3.1 presents a gradient of static analysis approaches from quantifying the quality of source code using patterns and heuristics, to verifying specific properties by leveraging language semantics. Section 3.2 presents the most used family of dynamic analyses, aka testing. It introduces software testing practices in general and presents dynamic-analysis approaches leveraging tests. Finally, Section 3.3 presents crosscutting approaches focusing on reducing resources needed to analyze and test source code without reducing precision and recall. The application of these technics to the analysis of source

code histories will be detailed in chapter 5.

Definition 2 – Impact: Effects of evolution on program properties. Source code evolution can affect observable properties of the systems under study, be it from compilation, linting, tests, or benchmarks. Given P a property, P state can either be pass or fail. We classify code evolution effects into three categories, namely:

— *Impacting Evolution:* P goes from pass to fail, namely the evolution breaks P

— *Repairing Evolution:* P goes from fail to pass, namely the evolution repairs P

— *Effectless/Neutral Evolution:* P state does not change

Definition 3 – Dependency graph. A graph where nodes are elements of an Abstract Syntax Tree (AST) and relations are representing dependencies between AST elements. Here dependency means that when changed or broken, it might have effects on properties of the program represented by the AST as defined in definition 2. There are multiple types of dependency relations.

For the semantic of a programming language, we consider:

— **Referential relations** between a declaration and its references, such as between: a callee and a caller, a variable/attribute access and declaration, a type reference and a type declaration.

— **Structural relations** between a parent and children AST elements, such as between a class and its methods, a type and its inner/associated types.

— **Data relations** *i.e.,* data dependence (related to race conditions) between a read and a write.

— **Package dependencies** between a library and its usage in an app or another library. A package manager handles these dependencies either at a source level or at the system level.

A dependency graph can take the form of an overlay graph, and association table, or simply attributes of AST nodes referencing other AST nodes.

Static and dynamic analyzes can extract different relations for representing dependencies among elements of an AST. There are multiple well known structures and approaches that can be part of a dependency graph, in particular:

— a *call graph* relates method/constructor/lambda declarations to calls;

— a *dataflow graph* typically relates variables writes to variable reads and in general the flow of data through languages constructs;

— a *type hierarchy* relates type declarations through inheritance;

— a *typed AST* has types assigned each expression;

— a *scope graph* relates references and declarations through scopes [Ant+18].

— a *package dependency tree* relates packages with the help of a package manager *e.g.,* `npm ls --all` or `mvn dependency:tree`.

3.1 Static analysis

In the context of this thesis, a number of static analysis are needed to scale to large software systems. Indeed, dynamic analysis requires building and executing the analyzed program, becoming impractical for large projects. In comparison, static analysis can be adapted indifferently from runtime to infer a lot of results that could be obtained through dynamic analysis.

Multiple kinds of semantics can be distinguished: integers, floats, pointers (memory), references, types, etc. These semantics depend on each other to fully analyze a program, for example, `x/(1-1)` can easily be checked for illegal operations just using semantic of integers, but once we have `int a = 1-1; x/a`, it is necessary to resolve references, replacing the reference to `a` by its value. It is also common to analyze source code by matching code patterns. In general, a static analysis without an extensive knowledge of data flow are considered more like a structural analysis. It is a notable distinction due also to the cost and complexity of computing such information.

3.1.1 Structural analysis

Structural analyses can be further divided by the amount of syntactic information they consider. Indeed, some basic features can be provided by considering source code as any other text, similar to what *Git* does to store source code histories (see section 2.1.2). It has the advantage of limiting complexity and specificity of the tools while enabling the use of existing text based tools.

Exact and fuzzy code search can be achieved with text based approaches. For instance, to search and navigate source code, source code forges (*e.g., GitHub*, *Gitlab*) use of-the-shelf text indexing tools, such as *ElasticSearch*. Yet these of-the-shelf tools have limitations. In 2022, *GitHub* replaced *ElasticSearch* with its own code search engine [Gita], that uses text heuristics to optimize indexing. Additionally, Blobs are classified (and searchable) by language using a number of heuristics.[1]

Ctags is a legacy tool used to extract declarations, such as functions, variables, and types. It notably provides an index of declarations to *Cscope* (see next section).

Palomba et al. [Pal+17] predicted bug-prowness of *Java* classes by learning composed thresholds on specific static code metrics.

Cocinelle [LM18] introduced by Lawall and Muller allows to match and apply code patterns (they call *semantic patches*) in C source code. It performs best-effort type inference, but no alias analysis nor dataflow analysis (it has to be implemented in the *semantic patches*).

[1] `https://github.com/github-linguist/linguist`

3.1.2 Semantic analysis

Semantic analyzes use semantic information about the program and language under analysis to simulate its behavior. Semantic analyzes often manipulate an intermediate representation to facilitate reasoning, notably the flow of data, *i.e.,* data flow analysis. However, producing such intermediate representation is non-trivial and might require to consider a significant amount of semantic information, such as with name resolution.

The semantic of references (also called name resolution) is an integral part of compilers and static analyzers, it is used to resolve variable accesses, function calls, and type references. Another major application of this semantic is the enhanced visualization and navigation of source code, often in the form of call graphs and type hierarchies. The Language Server Protocol (LSP) specification [Mic] defines methods to navigate referential relations, such as `references, definition, declaration, typeDefinition, implementation`.

While in compiler and static analyzers, references must be resolved unambiguously, it is not mandatory for human consumption. Indeed, depending on the language, the reference analysis can be a non-trivial task. For example in *Java*, it is possible to shadow,[2] obscure[2], hide,[3] overload[3], or override[3] a declaration. Thus, the analysis of partial *Java* source code is often ambiguous [DH08].

Nonetheless, Dagenais and Hendren [DH08] statically analyzed partial *Java* programs, to infer type facts, considering ambiguous references.

In comparison, *Cscope* has worked fine for decades, with the *C* language having a simple semantic of references.

GitHub has put many efforts adding code navigation through reference in their web interface. They particularly worked on a project named Semantic [Git22] to transform ASTs using compositions of semantic rules. It notably relies on Tree-Sitter [Mic22] to obtain Concrete Syntax Trees (CSTs) and fused-effects to compose semantics. Yet they had reliability issues with directly composing entire semantics, so they feel back to using an approach focussed on name resolution. Indeed, Semantic is now using stack graphs [4] [CA22] itself based on scope graphs [Ant+18]. With scope graphs, Antwerpen et al. presented an approach representing referential relations in scopes with a graph. For a specific class of referential semantics, this formalization enables the resolution of local referential relations in partial source code. Poulsen, Zwaan, and Hübner are working on extending the applicability of this approach to handle languages, such as *Java*, they call multi-phased name resolution [PZH23]. With *stack-graphs*, Creager and Antwerpen implement and extend scope graphs to also resolve members accesses, *i.e.,* first resolving the object type before resolving the member.

[Soe+16] do test selection, including dynamic bindings (overestimation).

[2] `https://docs.oracle.com/javase/specs/jls/se8/html/jls-6.html#jls-6.4`
[3] `https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#d5e13748`
[4] `https://github.blog/2021-12-09-introducing-stack-graphs/`

Wu, Zhu, and Li [WZL21] associate a call graphs with evolutions.

Example of shadowing in Java:

```
class A extends B { C x; } class B {} class C {}
```

Given this *Java* source code, adding an inner class `C` to `B` changes the reference to `C` in `A`, effectively hiding the outer class `C` from inside `A`. The type hierarchy takes priority over the structural hierarchy. In practice, to unambiguously resolve a type reference in a *Java* class, all its parent types definitions must be known.

```
class A extends B { C x; } class B {class C {}} class C {}
```

3.2 Dynamic Analysis

Dynamic analysis focuses on the concrete behavior of software. Compared to static analysis, functionally, it has the possibility to access to runtime context, thus it is particularly useful to observe concrete behavioral changes, *e.g.,* through regression testing. The concreteness is also a challenge as it directly limits the coverage of behaviors requiring multiple runs. However, dynamic analysis can be laborious due to issues tracing back to source code, while additional debug information and instrumentation will decrease runtime performances.

In the context of this thesis, by relying on the build system and runtime intended to be used with the software projects under analysis, dynamic analysis is a robust way of observing consequences of changes. Indeed, I used test executions for chapter 5 to extract information about the effects of code evolutions (definition 5), which directly participated to confirm the finding of co-evolutions (definition 6).

The section presents dynamic analysis and notably focuses on testing (non-exhaustive overview). Indeed, dynamic analyses are often relying on tests because they aim to cover as much as possible the possible behaviors expressed by the program, in a controlled environment. The frontier between testing and dynamic analysis is often "fuzzy". Multiple dynamic analysis approaches target tests, such as test coverage, mutation testing, test synthesis, fuzzing, parametric testing. Finally, we will also see non-functional analyses, *i.e.,* to measure performances of a program, as dynamic analysis is often the only way to do so.

3.2.1 Testing

Software testing is a software engineering practice that aims at evaluating an aspect of a software system to check whether it meets the expected result [HH88]. Software testing takes a large place in the current development process: developers manually write most of

```
1  export class Counter {
2    constructor(private x: number)
     ↪ {}
3
4    count(cb?: (n:number)=>number)
5     : number {
6      if(cb === undefined) {
7        this.x++;
8      } else {
9        this.x = cb(this.x);
10     }
11     return this.x;
12   }
13 }
```

```
1  test('count with undefined fct',
   ↪  () => {
2    const c = new Counter(0);
3    expect(c.count()).toBe(1);
4  });
5
6  test('count with fct', () => {
7    const c = new Counter(3);
8    expect(c.count(x => x-2))
9      .toBe(1);
10 });
```

Listing 1 – an object-oriented class (left) and two associated tests (right)

the tests, which can be time-consuming [Bel+17]; test suites, developed and maintained in parallel to software code to assess the correct behavior of this last, can be large [Zai+08]. A test suite is composed of a set of tests, *i.e.,* executable scenarios, that check the correctness of a given element of the software. Listing 1 depicts a basic example of code and two associated tests. The class *Counter*, on the left, has one private class attribute x, a public constructor that sets this attribute, and a public method *count*. The two tests, on the right, that test different usage scenarios of the method *count*: The first test calls the *count* method without any argument. It thus checks that x is incremented (line 7 in the class code). The second test calls the *count* method with a function as argument. The x attribute thus takes the result of this function with x as argument (line 9 in the class code).

There exists different types of tests that work at different granularities, and that use different technics to achieve their goal. For example, the two tests of Listing 1 are *unit tests*: they focus on testing different usages of a unit (here the class *Counter*) without any dependency to other units (*i.e.,* other classes). The goal of unit testing is to check that an isolated unit works as expected before testing in interaction with other units. It often requires mock implementations to tests units in isolation.

When it is difficult to isolate units, another kind of tests is *end-two-end (e2e) testing*: an *e2e* test simulates a user that interacts with the system to perform a specific usage scenario of the system under test. This is often related to *system testing* as it can sometimes work on the whole application under a specific scenario instead of a code unit. This is also related to *acceptance testing* that consists of e2e / system tests described in the specifications of the system and defined by the client. The goal of acceptance testing is to provide the client with tests that demonstrate that features of the system work as expected. An especially effective approach when a large amount of input can be used along

concise oracles/properties is property-based testing where inputs are generated instead of being concretely written.

When it is difficult to specify oracles, it is possible to fall back to catching regressions (*i.e., regression testing*) with *snapshot testing* where previous execution outputs are saved then comparing to new ones. With the drawback that it might also catch improvements. When even storing test outputs is problematic, it is still possible to use *fuzz testing* to assess a system's robustness by trying to provoke crashes while providing random inputs.

To cover many complex features, the number of tests (including along generated inputs) can become very large, hence, taking a long time to execute along with a lot of resources. The major trade-off resides in the balance between catching regressions, obtaining quick feedbacks from the test suite, and maintaining tests (when there are changes to features or dependencies).

3.2.2 Approaches Synergizing with Tests

Beyond its original goal of validating the behavior of a system under test, software engineering techniques and research works leverage test suites to achieve various purposes, including the improvement of said tests. Indeed, tests suites execute software system code so that one can extract run-time information from these executions. Various techniques used such run-time information, in particular to propose software testing techniques that complement tests. *Code coverage* is a testing assement technique that uses test execution information to mark code lines or code statements as covered or not by at least one test [ZHM97]. This technique gives as output a metric to evaluate the quality of a test suite and helps developers in improving it. For example with Listing 1, the execution of the two tests give a coverage score of 100% of the code statements of class *Count*: the two tests cover all the code statements of this class. This, however, does not mean that the code is safe, as we will see in the next paragraph.

Complementary to test coverage, *mutation analysis* is another metric that leverages test execution information [JH10; Pap+19]. At run time, mutation operators change specific code instructions (for example, replace a `+` instruction by a `-` one) to check whether the test catches the mutant, *i.e.,* the fictive alteration of the code. These two metrics, test coverage and mutation score, aim at providing testers with metrics to assess and improve the quality of their test suite.

In response to bad coverage or low mutation score, automated approaches can be used.

Test amplification *"consists of exploiting the knowledge of a large number of test cases, in which developers embed meaningful input data and expected properties in the form of oracles, in order to enhance these manually written tests with respect to an engineering goal"* [Dan+19].

[KZ19] combine multiple testing approaches to improve performances of parametric

testing on system tests. In short, this approach extract unit tests from system tests, then executes the extracted unit tests with parametric technics, finally, it lifts back failing tests to system tests to validate their verdicts.

For certain programs that are inherently complex to test well in a limited number of tests, *i.e.,* requiring to use many inputs (but respect notable invariants), such as parsers and protocols with possibilities of deadlock, it is important to first focus and leverage parametric approaches, such as *Quickcheck*, and employ delta debugging technics to quickly simplify failure-inducing inputs [ZH02].

Commit slicing [Li+17; LRC22] is also a notable use of tests, leveraging test coverage to approximate code connexity. Actually, a thorough code coverage is necessary to infer proper dependencies between code elements and (consequently) changes.

3.3 Hybrid analysis

In the context of this thesis, scaling code analysis technics is crucial to reach developers that work on large projects. This section presents how this scaling problematic is tackled in the literature. Indeed, static and dynamic analysis can be combined to leverage the efficiency and exhaustiveness of static analysis together with the precision and concreteness of dynamic analysis.

Damodaran et al. [Dam+17] present an overview of malware detection approaches and the benefits of hybrid analysis, specifically on performance concerns.

Test selection allow to reduce the number of tests that need to be executed [ERS10; Soe+16].

Memoization/Caching of intermediary build artifacts makes builds incremental and reduce built time [SA93], it can be done using *Ccache* with *make*, some build systems are designed to build incrementally large code bases, such as *Buck2* (*Meta*) and *Bazel* (internally *Blaze* at *Google*).

With *JUnit*, once tests are built it is possible to execute them in parallel, reducing the duration of the test suite's execution.

3.4 Conclusion

This second background chapter presented another dimension of source code analysis, this time through "space" (in opposition to "time"), *i.e.,* analyzing relations between code elements at a given state (version/commit). Indeed, to concretely relate evolutions we defined the effect of a change on other code elements and properties of a program, we also defined the dependency graph that enables to related code elements through different dependency relations and propagate effects of a change, *i.e.,* it should allow to model and predict the effects of changes throughout dependency relations (see chapter 5).

Approaches from the state-of-the-art of source code analysis were presented through the different categories of static, dynamic and hybrid analyzes. Moreover, as an essential component of advanced analyses, it specifically focussed on reference analysis and name resolution, as it is fundamental component of code navigation, many advanced source code analysis, building of dependency graphs (definition 3), and predicting impact of changes (definition 2). Finally, to scale to real world problems, it emphasized the performance concerns of source code analysis, through different approaches, such as incrementally, and parallelization.

Chapter 4

Temporal Code Analysis and Analyzing Evolving Software

> "Within Google, we sometimes say, 'Software engineering is programming integrated over time.' "
>
> — Titus Winters *et al. Software Engineering at Google (2020), p. 3*

This chapter presents the state-of-the-art on how temporal analysis and spatial analysis of source code are combined, with the aim of better understanding software engineering practices and automating software engineering tasks. It starts with a classification of temporal code analyzes (section 4.1) that allow us to better understand the problems faced by researchers and their needs when analyzing source code histories. Then section 4.2 presents an in-depth state-of-the-art on source code co-evolution.

4.1 Classification of temporal code analyzes

Temporal code analysis refers to a code analysis that studies the code of a software system over several moments of its existence. Such a class of analyzes differ from history-based analyzes in the way that one can study the different commits of a history with no interest in temporality or code evolution. So, not all history-based analyzes are temporal code analyzes. For example, Herbold et al. [Her+22] analyzed tangled history commits singly with no focus on temporality. Similarly, Śliwerski, Zimmermann, and Zeller [ŚZZ05] proposed an approach for analyzing history commits to state whether a given commit induces bug fix. Fischer, Pinzger, and Gall [FPG03] proposed an approach for building a release history database by merging information from code histories and bug tracking systems. The authors then used this approach to study change couplings [FGP05]. Instead, a temporal code analysis may study the evolution of code metrics over time [Bal+97; Rap+04] or track the (co-)evolutions of code elements [Gru+21; HHK20; Zai+11; Le +21]. The interested readers can refer to the survey of Kagdi, Collard, and Maletic [KCM07] for more details about approaches that mine software repositories and therefore their history.

Here, we propose to classify temporal code analyzes into three main categories, namely *batch*, *tracking* and *cross-AST*.

4.1.1 Batch temporal code analysis

The *batch* category corresponds to analyzes that operate on several versions (*e.g.,* several commits) sequentially without requiring relations between the underlying ASTs of those versions. The approach of Tufano et al. [Tuf+17], that studied how smells appeared in source code, typifies this class: the approach checkouts each commit sequentially to launch a code analysis on it and get its results.

We identified four sub-categories separated into two groups: *external or internal batch analyzes* and *semantic or textual batch analyzes*. *External* batch analyzes refer to the use of an external standalone tool applied on each version (*e.g.,* Decor [Moh+09]), such as in [Tuf+17; Pal+18] to detect bad smells in Java files. *Internal* batch analyzes refer to ad hoc analyzes specifically developed for the approach. For example Rapu et al. [Rap+04] studied the stability of classes by launching a specific code analysis on each studied version of software systems in a batch process. The analysis of one version produced metrics for all the classes of this version. The process is then repeated for different versions to study the stability of classes. Gall, Hajek, and Jazayeri [GHJ98] studied the evolutionary coupling between files (*i.e.,* which file changed with which other file). Alexandru et al. [Ale+19] computed the cyclomatic complexity of classes over versions. Batch analyzes may require syntactic knowledge of the language used in stored files [Zim+05]. We call this sub-category *semantic* batch code analyzes. For example, studying the stability of a method requires to locate methods in files based on the used language. Textual batch code analyzes do not consider such semantic. For example, studying the evolutionary coupling between files does not require any syntactic knowledge.

4.1.2 Tracking temporal code analysis

The *tracking* class of analyzes is close to the *batch* class in the way that the approaches analyze code sequentially but with the challenge of identifying the new position of a code element (*e.g.,* method, statement) in the different versions. The challenge in such analyzes is that tracking a code element requires both syntactic knowledge of the language used in stored files, and a dedicated heuristic to follow an element as precisely as possible. For example, tracking a method requires knowing what corresponds to its beginning (*e.g.,* annotations or a *JavaDoc* block for *Java* code) and its end. To bypass this requirement in order to move from the *tracking* to the *batch* category, several approaches made a simplifying assumption. For example, Khomh et al. [Kho+12] considered that one *Java* class corresponds to one *Java* file, thus enabling the use of batch process without tracking precisely *Java* classes.

Several approaches track code elements using, for example, meta-programming tools, such as Spoon [Paw+15] using Git commands or Git-based tools that eases the analysis of Git histories (*e.g., PyDriller* [SAB18]). For example, the Git command *log* has an option

L to track given lines in a given file, as used in [LH22; Blo+18]. Examples of tracking are: using Git commands to compute the change- and error-proneness of code chunks [Blo+18] or to study the evolution of methods complexity [LH22]; techniques for getting method histories, such as *CodeShovel* [Gru+21] or *FinerGit* [HHK20]; understand when a null pointer issue or a bad smell appeared in a commit [Cor+16];

4.1.3 Cross-AST code analysis

The *cross-AST* category corresponds to analyzes that operate on several ASTs at the same time to investigate complex relationships between code evolutions. For example, studying the co-evolution process of code and their tests [LY17; Le +21] or identifying causes for bugs code repair [Lin+07; Ni+20].

Wu, Zhu, and Li [WZL21; WLL21] combine call graphs with evolutions computed by *ChangeDistitiller* [Flu+07] in a database, it is presented as a formalization of internals of *CSlicer* [Li+17].

Padioleau, Lawall, and Muller [PLM06] presents evolutions to Linux kernel drivers, they call collateral evolution, that occur in response to evolutions in the kernel and driver support libraries.

A refactoring [TKD20; Sil+20] can also be considered as a combination of causally related evolutions that can be fully handled statically.

To summarize, performing temporal code analyzes, *i.e.,* analyzing the AST of the same program at different times, faces the following issues:

— For cross-AST analyzes, stakeholders have to write and maintain *boilerplate glue code* to put in relation multiple ASTs, such as in [Le +21];

— As a consequence of the previous point, analyzes may *consider evolutions at a coarse granularity,* i.e., *the class or method level only*, such as in [Pal+14], while temporal analyzes might require more fine-grained evolutions (*e.g.,* instructions);

— Analyzes may face *memory and CPU over-consumption* when several ASTs have to be compared or analyzed together (*i.e.,* not a batch process), thus preventing them to scale, as detailed in [Le +21];

— Stakeholders have to *write and maintain their own API for performing temporal queries*, *i.e.,* queries on several versions of an ASTs (*e.g.,* asking the stability of a given class [Rap+04]);

— Stakeholders have to *re-analyze the program history on every new analysis*;

— Current temporal code analysis approaches (*e.g.,* [Ale+19]) support the computation of syntactical metrics in a sequential batch way as a temporal code analysis.

4.2 State-of-the-art on Software Co-evolution

This section presents an overview of the literature about software co-evolution, that aim to show how existing approaches combine temporal and spatial analysis to automate source code maintenance, notably tests. Section 4.2.1 first establishes the methodology and comparison criteria to help organize existing research works on software co-evolution. Then section 4.2.2 presents and details the resulting classification.

4.2.1 Methodology

This section presents the methodology I used to build this state of the art. I propose criteria to categorize approaches that work on code and test co-evolution. Thanks to those criteria I propose to classify the literature and choose best-suited techniques depending on particular concerns. Figure 4.1 illustrates those criteria as a feature model.

The comparison criteria are inspired from the survey written by Hebig *et al.* [HKB16] on the co-evolution of models. To gather relevant papers, the bibliographical research started with a set of articles provided by my supervisors. Then I searched on mainly *google scholar* with keywords from previous papers and also followed the most relevant references from papers that I read, known as the *snowballing technique*.

In this classification, co-evolution is considered as a two-step process, where the first step would be about evolution, while the second step would be about co-evolution. Both steps contain specific categorization criteria that will now be detailed along with the degree of automation and language characteristics.

Degree of automation

One of the first criterion to consider is the **degree of automation** of the co-evolution. It quantifies the amount of involvement needed by a developer in the process of co-evolution. In case of a full automation one might only have to confirm co-evolution, otherwise in a semi-automated co-evolution one might need to choose between possible resolutions to apply or even create a custom transformation, capable of handling some domain-specific evolutions. We distinguish *manual*, *semi-automated* and *fully-automated* approaches.

Language characteristics

The software systems that can be co-evolved contain different characteristics. Those characteristics can particularly be observed through the programming language point of view. Most software projects use various frameworks and programming languages. This multitude of languages might share common characteristics. We mainly consider: *i)* the language paradigm, i.e., *Object oriented*, *Imperative* or *Declarative* and *ii)* the type system, i.e., *strongly* or *weakly* typed languages.

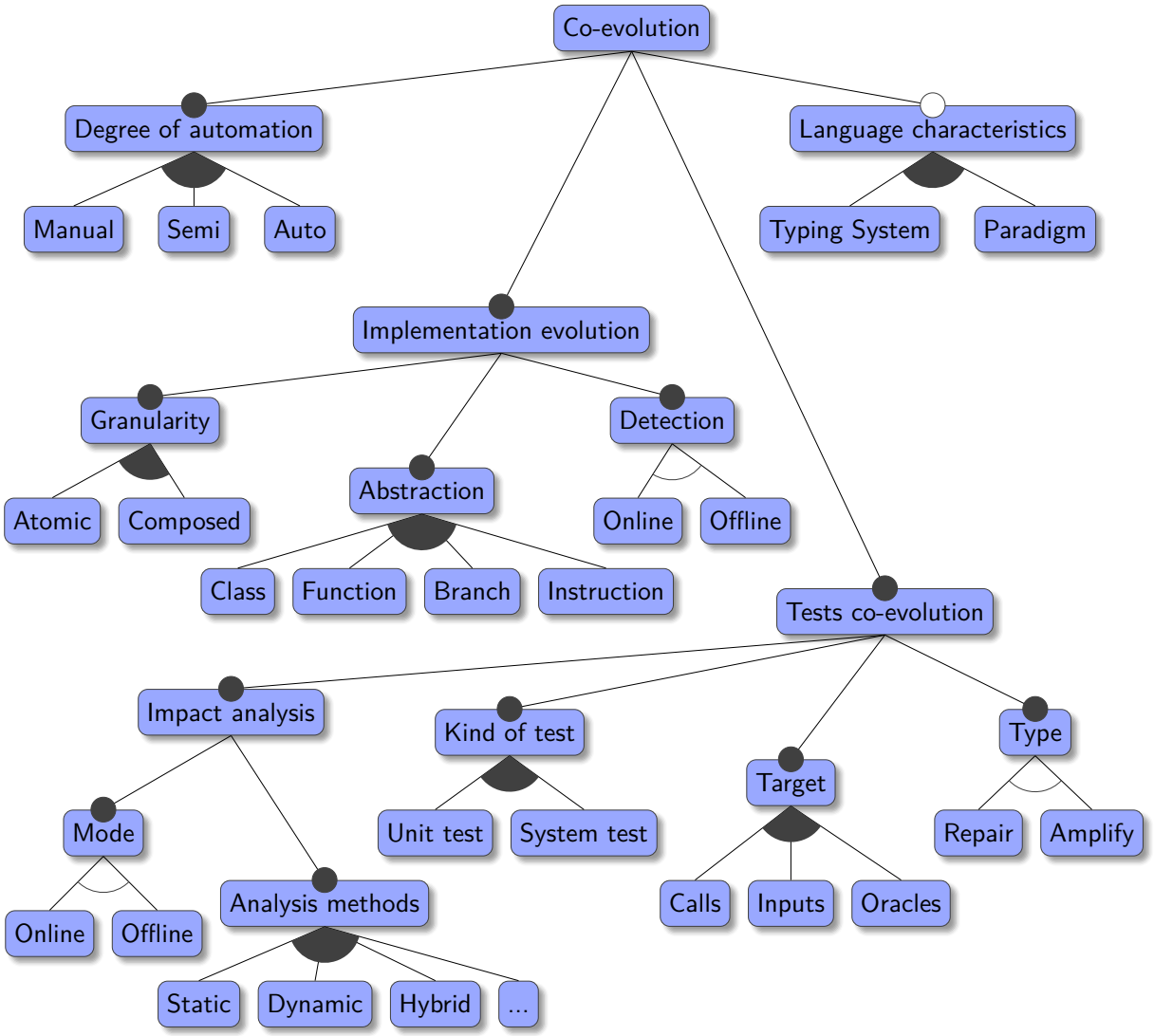


Figure 4.1 – Feature model of co-evolution of code and test [▲ : or, △ : xor, ● : mandatory, ○ : optional]

Detection and classification of evolutions

Detecting and classifying evolutions is the first step before considering co-evolution. Each major criterion part of this step of co-evolution are explained in the following:

1. Granularity: The **granularity of evolution** is important to the automation of the co-evolution. The simplest kind of evolutions is an *atomic* change, while it is very simple to detect simple changes, they contain little information alone. Additions, deletions, and modifications are the simplest atomic changes, and often the only atomic changes considered. Moreover, it is possible to combine atomic changes into *composed* changes that bring additional contextual information. For example, moving a method from one class to another is composed of a deletion and addition of method.

2. Level of abstraction: In every software analysis, the **level of abstraction** reflects the

trade-off made between precision and performance. For example, the *file* abstraction can be considered as high abstraction to detect changes in a code. The file abstraction is what most compilers for procedural languages are using to avoid recompiling unchanged files. There is also the *class* abstraction, it is one of the most used, as it syntactically and statically presents a large quantity of semantic information. In fact, methods are carrying the behaviors of object, and behaviors can be shared through inheritance. But this abstraction requires the analyzed language to be object-oriented and possibly have class, prototypes and an inheritance system. To establish measurements of impacts from changes it is necessary to look at calls, this abstraction is a *call graph*. Finally looking at the level of *flow graphs*, i.e., blocks of instructions linked by branches might be necessary for some analyses, but it requires a lot of effort and processing power to compute.

3. Detection: The **detection of changes** can be done *online* by logging operations made on files or *offline* by comparing states of files between versions. Detecting changes through *online* logging is more precise but is also more intrusive than *offline* detection. *Online* detection can be brittle in case of unlogged changes. Thus, all external tools modifying the code would need to provide the set of applied changes.

Co-evolution of tests

Here, we will look at the particular aspects that concern the actual the co-evolution of code and test.

1. Impact analysis: The **impact analysis** of code changes on tests need to be quantified to propose relevant co-evolution. It allows to locate tests that need to be co-evolved and to provide some more contextual information on tests dependencies. Two modes of impact analysis can be discerned. *Offline* impact analysis is computed when the developer is done with his current set of changes. While *online* impact analysis is computed interactively whenever a change happens.
   Moreover, several methods can be used herein depending on the type of language used for the code. We can distinguish *static analysis* for statically typed code, *dynamic analysis* for dynamically typed code, or *hybrid analysis* too.
2. Kind of tests: The **kind of tests** targeted by a tests' co-evolution methods could be relevant as *system tests* are much bigger and take longer than *unit tests*. In a way the kind of tests handled by co-evolution methods should give a lead on the scalability of the approach.
3. Type: Given some evolutions, two **types of co-evolution** are possible. *Amplification* co-evolution creates new tests from other tests by various exploratory methods (genetics, regression, etc.). *Repair* co-evolution modifies existing tests to make it pass the compilation, or the runtime checks.

4. Target: The **target** of the test co-evolution can be the *calls*, the *inputs* of calls, the expression of *oracles*, or a combination of the 3.

   Calls in a unit tests refer to invoking specific functions or methods that are being tested. This is essentially the action you want to test and evaluate. In other words, a call is the execution of the code you want to verify.

   Inputs refer to the data or parameters that you provide to the function or method being tested. These inputs are crucial for assessing the behavior and correctness of the code under different scenarios. From a list of call it is possible to use the corresponding declarations to infer possible parameter values.

   Oracles set the standard of correctness for the program under test, *i.e.,* oracles enforce a concrete program specification. In tests, oracles take the form of assertion statements.

   For example, in the first test case of Listing 1, a *call* to the constructor of `Counter` is made with the `number` 3 as an *input*, then a *call* to the method `count` is made with a function as an *input*. Finally the *oracle* checks that the value return by the previous call is equal to the number 1.

Table 4.1 – Classification part 1, evolution of production code [?: not mentioned, N/A: not attributable]

Reference			Language	Granularity	Abstraction	Detection	Automation
Čubranić and Murphy	2003	[ČM03]	all	?	metadata, ...	offline	auto
Adamapoulos *et al.*	2004	[AHH04]	Fortan-77	N/A	mutant	offline	auto
Jiang *et al.*	2006	[Jia+06]	N/A	composed	event	offline	auto
Halfond *et al.*	2008	[HO08]	java, PHP, http,...	composed	calls, data flow	offline	semi [1]
Memon *et al.*	2008	[Mem08]	all	composed	event	offline	semi [2]
Hassan	2009	[Has09]	C,C++	atomic	pattern, metadata	offline	auto
Daniel *et al.*	2010	[DGM10]	*Java*, .NET	composed [3]	instruction	offline	semi[1]
Dagenais *et al.*	2011	[DR11]	*Java*	composed[3]	metadata	offline	semi[1]
Jin & Orso	2012	[JO12]	C	composed	class, flow graph	offline	auto
Mirzaaghaei *et al.*	2014	[MPP14]	*Java*	atomic	class	offline	auto
Dagenais *et al.*	2014	[DR14]	*Java*	composed	pattern	offline	semi[1]
Khelladi *et al.*	2017	[Khe+17a]	OCL	composed	class	online	semi[2]
Fluri *et al.*	2009	[Flu+07; GFP09]	*Java*	composed	instruction, class	offline	auto
Falleri *et al.*	2014	[Fal+14]	*Java*	atomic	instruction, class	offline	auto
Silva *et al.*	2020	[Sil+20]	*Java*	composed	instruction, class	offline	auto
Tsantalis *et al.*	2018	[Tsa+18b]	*Java*	composed[3]	instruction, class	offline	auto

[1]Makes recommendations, on possible co-evolutions.
[2]Might sometimes require human design choices.
[3]Only consider in place compositions.

Table 4.2 – Classification part 2, co-evolution of tests [?: not mentioned, N/A: not attributable]

Reference			Language	Impact Analysis		Kind of test	Type	Target	Automation
				Mode	Method				
Adamapoulos *et al.*	2004	[AHH04]	Fortan-77	offline	static, genetic	unit	generate	tests inputs	auto
Halfond *et al.*	2008	[HO08]	*Java*, PHP, http,...	offline	static	?	repair	calls in general	semi[1]
Memon *et al.*	2008	[Mem08]	all	offline	dynamic	unit[5]	repair	whole tests	semi[2]
Thummalapenta *et al.*	2009	[Thu+09]	*Java*	offline	static	unit	generate	tests calls and parameters	semi[1]
Daniel *et al.*	2010	[DGM10]	*Java*, .NET	offline	static, symbolic	unit	repair	whole tests	semi[1]
Robinson *et al.*	2011	[Rob+11]	*Java*	offline	static	unit[5]	generate	whole tests	auto
Jin & Orso	2012	[JO12]	C	N/A[6]	dynamic	unit	generate	tests calls and inputs	auto
Galeotti *et al.*	2013	[GFA13]	*Java*	N/A[6]	static, symbolic	unit	generate	tests calls and inputs	auto
Tonella *et al.*	2014	[TTN14]	all	offline	dynamic	all	generate	tests event	semi[1]
Mirzaaghaei *et al.*	2014	[MPP14]	*Java*	offline	static	all	repair, amplify	whole tests	auto
Fraser *et al.*	2014	[FA14]	*Java*	N/A[6]	static	unit	generate	tests calls and inputs	auto
Mirshokraie *et al.*	2015	[MMP15]	js	offline	dynamic	unit	generate	whole tests	auto
Mirshokraie *et al.*	2016	[MMP16]	js	offline	dynamic	unit	generate	whole tests	auto
Khelladi *et al.*	2017	[Khe+17a]	OCL	offline	static	N/A[7]	repair	whole models and constraints	semi[2]
Kampmann & Zeller	2019	[KZ19]	web-sql-python-C stack	offline	dynamic	unit	generate	whole tests	auto

[5]In the context of regression testing.
[6]Do not use changes to generate tests.
[7]Co-evolve OCL constraints.

4.2.2 Classification of approaches

In this section, we will present the state of the art on co-evolution of code and tests following the classification given by the feature model in Figure 4.1. We will present some of our results in Table 4.1 regarding the classification of approaches that detect and classify evolution in software artifacts, mostly code. Then in Table 4.2, regarding the actual co-evolution of software artifacts, mostly tests. It should be noted that approaches mentioned in one table but not in the other, either only detect evolution or only improve tests without considering evolution.

Language characteristics

As shown in Tables 4.1 and 4.2, most of the approaches that I found focus on object-oriented programming languages. In particular, they use the *Class* construct and strong and static type systems, like Java, C#, and C++. These approaches seem to correlate strongly with techniques, such as static analysis and patterns recognition. Nonetheless, some approaches do not rely on particular characteristics of languages in themselves, like class and static types, but they rely on the run-time behavior of the program. These approaches use events at some points with dynamic analysis to produce behavioral models [JO12; KZ19; Mem08]. We also found a few approaches working on declarative constraints systems [Khe+17a], or on database language paradigms [ZED11; KZ19].

Degree of automation

In both Tables 4.1 and 4.2, the automation criterion refers to the approach in general and not only on evolution or co-evolution, as it was very difficult to distinguish both. We noticed that most approaches are automatic, but many offer semi-automation.

Evolution of the Implementation

1. Granularity: Table 4.1 shows a correlation between the granularity of changes and the automation of the corresponding approach, where approaches using composed changes require more manual intervention. The cause of this correlation seems to be that approaches using composed evolution are more complex although they can handle a greater variety of evolution.

   All approaches considering evolution use some degree of composed changes, Dagenais *et al.* use some basic compositions in [DR11; DR14]. For example, renaming is composed of the deletion of a name and the addition of a new one. Here the authors consider the case of in-place renaming, so it is easier to infer the relation between the deletion and addition.

Tsantalis *et al.* [Tsa+18b] detects high level refactoring embedded in RefactoringMiner.[4] Other tools focus on detecting evolution changes. Gumtree [Fal+14] considers only atomic changes, RefDiff [Sil+20] and ChangeDistiller [Flu+07; GFP09] consider complex changes but are less precise/performant than RefactoringMiner [Tsa+18b].

2. Abstraction: Both Tables 4.1 and 4.2 show that the abstraction of choice is the class construct.

   Some approaches make use of metadata to mine patterns in Content Versioning Systems (CVS). Zaidman *et al.* in [Zai+08; Zai+11] mine co-evolution patterns in SVN commits, while Martinez *et al.* in [MM19], RefactoringMiner [Tsa+18b], Gumtree [Fal+14] considers only atomic changes, RefDiff [Sil+20] and ChangeDistiller [Flu+07; GFP09] mine co-evolution patterns in git commits.

3. Detection: Table 4.1 shows that most of the approaches that we found use offline detection. These approaches deal with contents that can be edited in many ways, making it difficult to change each editing mode. Thus, these articles rely either on file metadata and file diffs to detect changes [MPP14; DGM10; HO08], metadata of CVS and blob differences [MM19; Has09; DR11; ČM03; Tsa+18b], or behavioral differences [Mem08; Jia+06].

Co-evolution of tests

We found 12 approaches explicitly focussing on tests co-evolution. We also found a few others approaches that do not exactly co-evolve tests but similar purpose artifacts or do not explicitly use the co-evolution term.

1. Impact Analysis:

   Analysis mode All the articles retained in the state of the art are doing offline impact analysis. There is therefore either no need for test co-evolution during code changes, or the current test co-evolution techniques are too expensive to react to each change.

   Analysis methods We found different methods of impact analysis.

   Static analysis is the most wide spread type of analysis here, as shown in Tables 4.1 and 4.2. The causes of this distribution seem to be due to the large amount of semantic and structural information available in strongly typed object-oriented languages, such as Java.

   On the contrary, **dynamic analysis** does not appear to be very common, in fact, dynamic analysis is particularly suitable for highly dynamic and weakly typed languages, such as Javascript and Perl. But it requires to go down to the runtime of the program which causes a performance penalty and an increase in

[4] `https://github.com/tsantalis/RefactoringMiner`

complexity. Nonetheless in [KZ19] Kampmann at *al.* use dynamic analysis to synthesize unit tests from system tests through the use of behavioral models and fsm inference algorithms.

Mirshokraie *et al.* in [MMP13] then in [MMP15; MMP16] combine dynamic analysis and **mutation testing** to improve tests of Javascript programs.

Hybrid analysis seem to be in many future works [And+17] but we did not find approaches explicitly claiming it in the context of co-evolution.

2. Kind of tests: We found no approaches claiming to be able to repair or generate system tests. So the hypothesis on the computational complexity of these approaches does not seem invalid.

   Kampmann *et al.* in [KZ19] use system tests to generate new unit tests. Mirshokraie *et al.* in [MMP16] also use system tests but in the from of GUI tests to generate new unit tests.

   Memon *et al.* in [Mem08] generate unit tests in the particular case of regression testing. Here the regression testing allows the creation of oracles from the program current behavior.

   Khelladi *et al.* [Khe+17a] do not co-evolve tests but a very close artifact. In fact, they co-evolve OCL, a declarative constraint language on models, such as class diagrams. Here specifying constraints is very similar to specifying oracles.

3. Target: Tonella *et al.* [TTN14] and Jiang *et al.* [Jia+06] use traces to construct a functional behavioral model of an application. Then they generate new tests as skeletons of **calls** from paths in the fsm. Halfond *et al.* [HO08] detect parameter mismatch in multi-languages systems. Dagenais *et al.* [DR11; DR14] recommend alternatives for broken calls and for general references. Fraser *et al.* [FA14] produce tests composed of calls and inputs from *Java* generics. Daniel *et al.* [DGM10] compute new **inputs** for tests that maximize coverage through symbolic execution. Adamopoulos *et al.* [AHH04] amplify inputs through mutation testing and genetic algorithms. Zhang *et al.* [ZED11] amplify tests for database systems through the use of symbolic execution and genetic algorithms. Table 4.2 shows that fully automated approaches generating (non regression) unit tests are not producing tests with **oracles**. Except with Kampmann *et al.*, and Mirshokraie *et al.* in [KZ19] and [MMP15; MMP16] where they borrow oracles from system tests (such as GUI tests) to generates unit tests. combine dynamic analysis and **mutation testing** to improve tests of Javascript programs. In facts, oracles are part of the application specification, thus there can not be automatically generated. To overcome this restriction, Mirzaaghaei *et al.* [MPP14] use oracles from other tests, Kampmann *et al.* [KZ19] run system tests with the same inputs as unit tests to reduce false positives triggered by oracles in unit tests (if an oracle from an unit test fails,

the corresponding system test should also fail). Khelladi *et al.* [Khe+17a] repair constraints which is very similar to repairing oracles.

4. Type: Robinson *et al.* [Rob+11] use static analysis to generate tests then mutation testing to refine generated tests. Kampmann *et al.* [KZ19] synthesize unit tests from system tests.

   Mirzaaghaei *et al.* [MPP14] amplify and repair unit tests using carefully handcrafted patterns that matches certain evolution. However creating these pattern can be tedious. A partial solution to this problem could come from Khelladi *et al.* in [KKE18], who are combining repairing rules to co-evolve models given changes in metamodels.

   Daniel *et al.* [DGM10] repair tests using symbolic execution, more specifically they focus on repairing string literals. Memon *et al.* [Mem08] repair regression GUI tests, more specifically they repair the sequence of GUI events, sometimes it needs manual interventions when the approach does not find an appropriate resolution.

4.3 Precising the Definition of Co-evolutions in their Time Dimension

In view of the extensive state-of-the-art, in both spatial and temporal dimensions of source code analysis, there might be practical opportunities to consider causal relation between evolutions to precisely analyze and understand source code histories. Indeed, multiple Cross-AST code analyzes consider causal relations between evolutions. Yet even the exact name of such causal evolution differs depending on the exact application context.

The **collateral evolution** from Padioleau, Lawall, and Muller [PLM06] explicitly consider the fixing of drivers caused by changes to the kernel and support libraries.

Migrations from Lamothe, Shang, and Chen [LSC18] focuses on changes between dependencies API that necessitates changes to application code.

Refactorings can be seen as a modification or creation of a declaration that necessitates fixing references of this declaration.

Co-evolution is a term used in multiple studies of source code histories with the aim of understanding the development process of practitioners [Zai+08; Zai+11; MRZ14; Flu+09; VP18]. For the development and study of models. Co-evolution is also used to qualify related changes in model driven developpement [Dem+16; Kus+15a; Kus+15b; Sch+15; Sch+15; HKB16; KSW18; Cic+08; DIP11; Khe+16; Khe+17b; Khe+17a; KKE18; Khe+20].

Sometimes **coupled-evolution** is also used [DIP13; Di +14], some other times co-change [Sil+15; SVM19].

In the remainder of this thesis, we choose to use co-evolution to describe causally

related evolutions. We propose definition 4 that aims to be applicable concretely in diverse contexts. Moreover, we also propose four derivative definitions that allow to qualify special configurations of co-evolutions (definitions 4.A to 4.D), with the aim of producing more precise results.

In biology with a more observational definition, a co-evolution corresponds to evolutions that takes place in different species due to the same environmental stimulus.

Definition 4 – Source code co-evolution. A source code co-evolution is composed of: an evolution that breaks a property, *i.e.,* it is the ***cause*** of the co-evolution; a second evolution that fixes a property broken by the ***cause***, *i.e.,* it is the ***repair*** of the co-evolution. In a timeline, the ***cause*** would be the first applied, followed by the ***repair***.

We distinguish four types of co-evolution: namely ***complete*** and ***partial***, that have their own definition of the ***cause*** and the ***repair***, and ***immediate*** and ***delayed***, that distinguish by when the ***cause*** and the ***repair*** append through time (*i.e.,* different commits).

Definition 4.A – Complete co-evolution. A **complete** co-evolution:
1/ consider a piece of code that satisfies a set of properties;
2/ then applies evolutions that break some properties;
3/ and finally repairs all the broken properties.

Definition 4.B – Partial co-evolution. A **partial** co-evolution corresponds to all the scenarios that differ from the definition of a **complete** co-evolution, such that a **partial** co-evolution:
1/ considers a piece of code that satisfies a set of properties;
2/ then applies evolutions that break some properties;
3/ and finally **repair**s some of the broken properties.

Partial co-evolutions relax criteria to obtain and study other interesting cases of co-evolutions. Naturally, a co-evolution might switch between partial and complete when considering different properties.

Definition 4.C – Immediate co-evolution. An **immediate** co-evolution corresponds to cases when both the **cause** and **repair** are located in the same commit.

Definition 4.D – Delayed co-evolution. A **delayed** co-evolution corresponds to cases when the **cause** or **repair** are spread over several commits.

Moreover, considering that we can have at some point (*i.e.,* in a commit) a **partial** co-evolution. Then, later this co-evolution might be completed by additional evolutions (if not completed at least repairing additional properties). Thus, making this new composition of evolutions a **delayed** co-evolution.

4.4 Conclusion

This chapter presented the implications of combining temporal and spatial analysis of source code with the aim of better understanding source code histories and toward automating engineering tasks, especially maintenance tasks. It first proposed a classification of temporal source code analysis integrating the state-of-the art approaches, then focused on the state-of-the-art of source code co-evolutions. This state-of-the-art enabled us to enunciate a general definition of source code co-evolution, based on properties that can be precisely instantiated. Chapter 5 will propose an approach to automatically find such co-evolution, with precision by leveraging test verdicts, and at scale by leveraging diffs and static impact analysis. Chapter 6 will specifically target limitations of current temporal source code analyzes by providing a framework for incremental analyzes of source code histories.

Part II

Contributions

> "I considered that a machine to execute the mere isolated operations of arithmetic, would be comparatively of little value, unless it were very easily set to do its work, and unless it executed not only accurately, but with great rapidity, whatever it was required to do."
> — Charles Babbage, *Passages from the Life of a Philosopher (1864), p. 43*

> "The purpose of software engineering is to control complexity, not to create it." — Jon L. Bentley, *Programming Pearls*

> "The battle for a fast system can be won or lost in specifying the problem it is to solve." — Jon L. Bentley, *Programming Pearls*

The four next chapters present my contributions toward better maintenance of software through co-evolutions:

Chapter 5 demonstrates the feasibility of finding precise co-evolutions in large source code histories, leveraging static impact analysis and tests to select possibly impacted tests and precisely observe functional effects of combination of changes.

Then I address the efficiency of computing fine-grained changes and their impacts from code histories:

Chapter 6 revisits how source code histories are represented and processed, leveraging both the structured nature of code, its stability through time, and locality of intermediate analyses. It led to an approach called *HyperAST*.

Chapter 7 presents an approach leveraging the *HyperAST* to compute referential dependencies incrementally.

Chapter 8 presents a novel code differencing technique, called *HyperDiff* that leverage the *HyperAST* to compare commits at scale.

Chapter 5

Temporal Analysis of Source Code Histories through Code-test Co-evolutions

As presented in the background on source code evolution (section 2.2) developers change their code base over time, to adapt to new requirements of their users or to changes in the environment. Fundamentally these evolutions stretch over three dimensions: Corrective, Adaptive, and Perfective. Concretely, developers often use feature addition, bug fix, performance optimization, and refactoring. Using these keywords (as part of the conventional commit specification) facilitates the reviewing process and allow to simplify the constitution of changelogs. Indeed, it allows to associate invariants on sets of evolutions while automatically checking them: a **bug fix** should not change the API, a **refactoring** should not change the behavior of the source code, and a **feature addition** should not break existing features. Refactorings are particularly interesting due to the way we test software, especially regression testing, since refactoring should not change the behavior of the program.

Based on the works of Zaidman et al. [Zai+08] and Levin and Yehudai [LY17] on co-evolutions between production code (abbreviated as code) and test code (abbreviated as tests), I propose to automatically find co-evolutions in industrial Java codebases. Precisely identifying effects of production code evolutions on tests is a real challenge. To find a code and test co-evolution, a practitioner indeed *has to find and isolate impacting groups of evolutions.* This practitioner can hardly do this task by hand since *it requires to identify and test each possible combination of evolutions,* i.e., *to compile code and run for each combination the tests to evaluate potential impacting and repairing effects.* Then, the practitioner *has to associate an impacting evolution to its possible repairing evolutions*, following the same process. For 10 evolutions in a single commit, this could imply 2^{10} combinations to evaluate.

This chapter presents an automated approach for precisely finding code and test co-evolutions in *Java* and *Maven* source code repositories. Our approach supports both atomic and complex evolutions, such as refactorings. It first performs a static analysis of Git histories and *Java* code to identify groups of interacting evolutions and tests. It then applies several heuristics to reduce the combinatorial explosion of candidate co-evolutions.

The approach finally narrows down these results with a dynamic analysis, based on test execution and compilation, which evaluates the effects of evolutions on test verdicts and compilation results. This dynamic analysis is one novelty of our approach that permits to qualify with high confidence the causal relation between two groups of evolutions in production and test code, *i.e.,* the cause and repair of a co-evolution. Moreover, as another novelty in contrast to [Zai+08; Zai+11; LZP09; LY17], we refine the definition of code and test co-evolution to distinguish two types, namely *complete* and *partial* co-evolutions. The former is when the co-evolved test passes, and the latter is when it does not, hence, missing some co-evolution ingredients.

We conducted an empirical study on a curated set of 45 open-source systems having Git histories. Our approach found 612 co-evolutions among which 500 are contained in single commits and 112 are scattered on two separated commits. In particular, our approach detected 202 complete co-evolutions for which it exhibits a precision of 100 % and an underestimated recall of 37.5 % in detecting the code and test co-evolutions. Our approach also spotted the different kinds of code and test co-evolution that researchers manually identified in previous work [Zai+08; Zai+11; LY17].

In summary, the contributions of this chapter are as follows:

— The first, to the best of our knowledge, automated approach for detecting tests and code co-evolutions in single commits (immediate) and in separated commits (delayed);

— We conducted an empirical study on 45 open-source software systems to evaluate the performance of our approach;

— The study confirms the existence of immediate and delayed co-evolutions. It also gives evidences and permits the novel characterization of *complete and partial co-evolutions*;

— We bundled the set of detected co-evolutions as a knowledge base for further investigation on this topic;

— We provide a comprehensive dataset for replicating the empirical study. `https://github.com/quentinLeDilavrec/ICSME2021`

The remainder of this chapter is organized as follows: Section 5.1 instantiate the definitions given previously where properties are tests. Section 5.2 details the approach. Section 5.3 presents the evaluation of the approach. Section 5.4 concludes the chapter.

The content and results of this chapter were published at ICSME 2021 [Le +21].

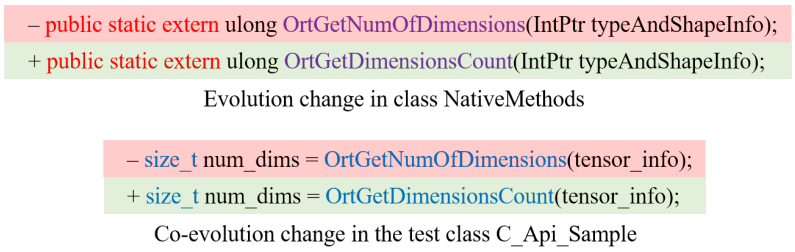


Figure 5.1 – Example of a *rename* change evolution and co-evolution.

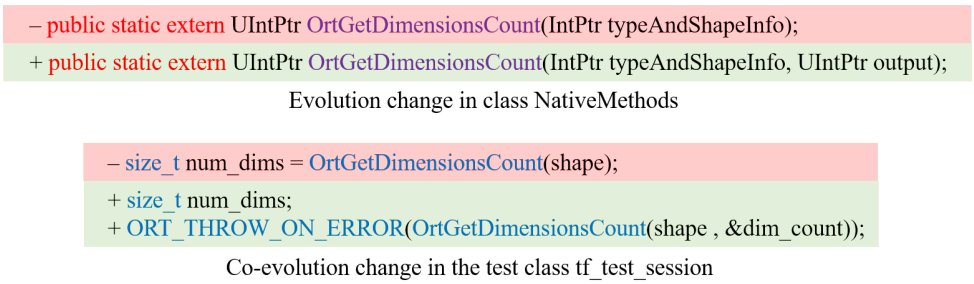


Figure 5.2 – Example of an *add parameter* change evolution and co-evolution.

5.1 Source Code Co-evolutions Instantiated to Tests

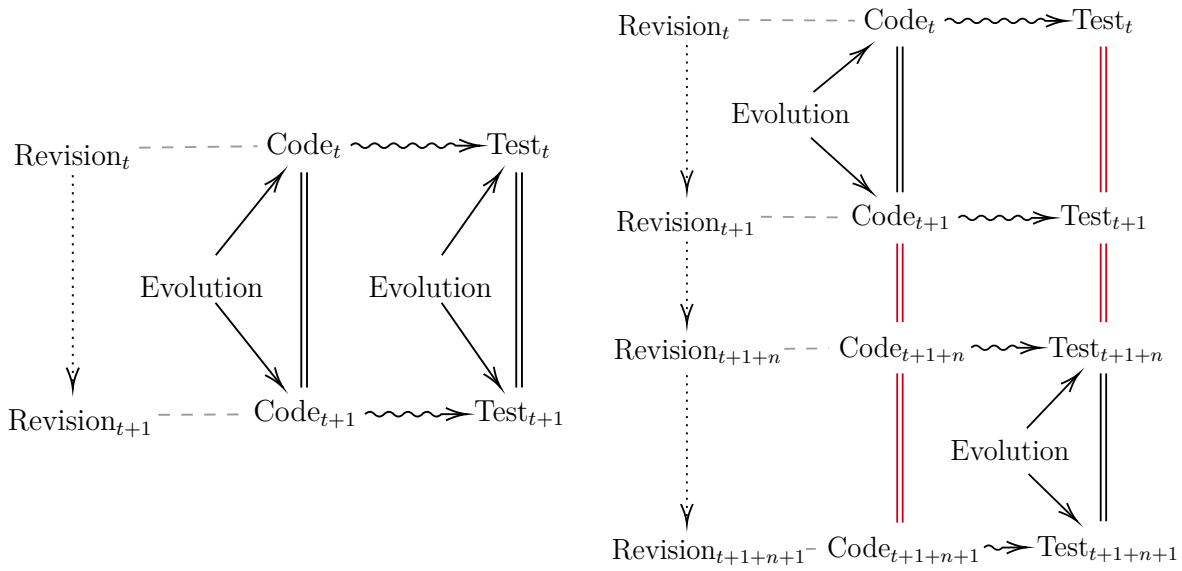
Given that we defined impacts and co-evolutions on abstract properties in definitions 2 and 4 at the end of chapter 4. To conform to code and test and observe effects and impacts these definitions must be instantiated considering more concrete properties. We choose three categories of properties: the **compilation** of the source/production code, the **compilation** of all tests and finally the **success** or **failure** of tests. These three categories have interactions. For *Java* codebase, due to build lifecycles, they are strongly dependent: to run tests all source code must compile, then for all tests to compile the whole source/production code must also compile. Any error short circuits the rest of the lifecycle. A code-test co-evolution denotes the idea of propagating the evolution of the code into tests, so they remain consistent with the modified code state.

Definition 5 – Effect of evolutions on tests. Code evolutions (on production code or on tests) can affect observable properties of the systems under study. In this work we use two properties: production and test code compilation (*i.e.,* a compilation passes or fails); and test execution verdicts (*i.e.,* a test **passes** or **fails**). We classify code evolution effects on these two properties into three categories, namely:

— *Impacting Evolution.* This is an evolution that now makes a test **fail** or prevents code to compile.

— *Repairing Evolution.* This is a code evolution that now makes a test **pass**. This implies that production and tests compilation **pass**.

— *Effectless/Neutral Evolution.* This is a code evolution that does not change the state of tests or compilation.



(a) Immediate (b) Delayed

Figure 5.3 – Co-evolution patterns [$\rightsquigarrow$: represents impacts, $\rightarrow$: represents the belonging of a piece of code to an evolution, $\cdots\!\succ$: represents the parent-child relation between two commits, $=$: represents the equivalence relation between pieces of code contained in different commits (it can be the identity relation, but also an evolution)]

To qualify the effects on compilation or on test execution, we rely on the following states:

1. **PCFAIL** *(level 1)*: production code compilation fails (test code not compiled or executed);
2. **TCFAIL** *(level 2)*: production code compilation passes but test code compilation fails;
3. **TFAIL** *(level 3)*: both compilations pass and the statically impacted test fail.
4. **TPASS** *(level 4)*: both compilations pass and the statically impacted test pass;

Definition 6 – Code and test co-evolution. A code and test co-evolution is composed of: a first set of evolutions in code that breaks tests, *i.e.,* it is the *cause* of the co-evolution; a second set of evolutions in code and/or tests that fixes tests broken by the *cause*, *i.e.,* it is the *repair* of the co-evolution. In a timeline, the *cause* is first applied, followed by the *repair*. We distinguish two types of co-evolution, namely *complete* and *partial*, that have their own definition of the *cause* and the *repair*.

Definition 6.A – Complete test co-evolutions. A complete co-evolution:
1/ starts at TPASS (before the cause);
2/ degrades to TFAIL, TCFAIL, or PCFAIL after the cause (and before the repair);
3/ restores back to TPASS after the repair (applied after the cause).

Definition 6.B – Test compiling co-evolutions. A partial co-evolution corresponds to all the scenarios that differ from starting to TPASS and ending to TPASS, and imply a degradation when applying the cause. For example, one partial co-evolution scenario is:
1/ it starts at TFAIL (before the cause);
2/ moves to TCFAIL after the cause (and before the repair);
3/ moves back to TFAIL after the repair (that does not strictly repair in such cases).

Figure 5.3a presents an immediate co-evolution in a code-test context, where both production code and tests evolve, and with the breaking and repairing evolutions in the same commit. Symmetrically, fig. 5.3b presents two groups of evolutions, yet they are part of different commits, possibly separated by many others, thus making it a delayed co-evolution.

5.2 Overall Approach

This section presents our approach for finding code and test co-evolutions from code histories. Figure 5.4 shows the overall process of our approach. After cloning the evolution history of a given project, it detects evolutions $\boxed{A}$. Then, a static analysis computes dependency graphs between evolutions in production code and test code $\boxed{B}$. Next the approach organizes evolutions $\boxed{C}$. Through a dynamic analysis, it then qualifies the effects of each group of evolutions on tests (as impacting, repairing, or neutral) $\boxed{D}$. Finally, the approach assembles the production code evolutions that *impact* test code, with the test code evolutions that *repair* tests $\boxed{E}$.

5.2.1 Detecting evolutions

The first step $\boxed{A}$ of the approach consists in detecting all the evolutions between two commits t and $t+n$, with $n \geq 1$. The approach relies on state-of-the-art techniques for detecting atomic and complex evolutions, respectively with *Gumtree* [Fal+14] and *RefatoringMiner* [Tsa+18a; TKD20].

5.2.2 Extracting dependency graphs

The second step $\boxed{B}$ of the approach consists in identifying evolutions in production code that *might* have impacts on tests. This step involves a static analysis that explores three kinds of dependency graphs (see Definition 3) from each identified evolution. For each evolution e located in the production code, the static analysis identifies the corresponding elements $e_0 ... e_n$ in the AST and extracts the three kinds of graphs from them. The construction of each graph stops when reaching an AST element located in test code.

Figure 5.5 illustrates the static analysis based on the example of Figure 2.4 and its evolution #1 that consists of a new parameter b in the declaration of the method *aMethod*. The static analysis starts from b to explore its type (*int*) and its method *aMethod*. The analysis then gathers all

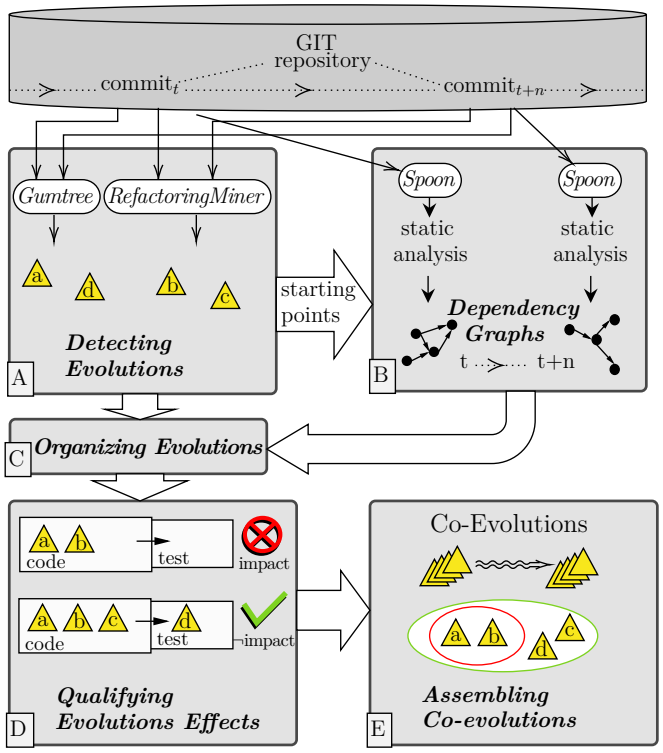


Figure 5.4 – Overall process of the approach.

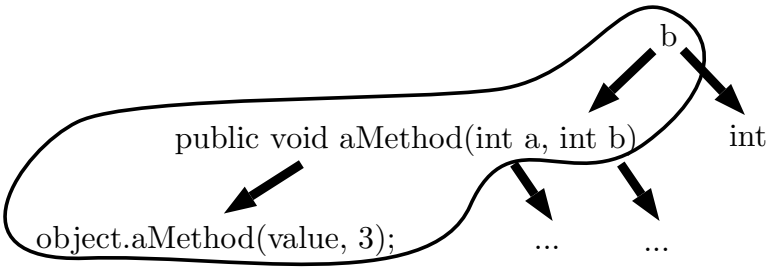


Figure 5.5 – Illustration of the static analysis, based on Figure 2.4. The bubble represents one dependency from an AST element of an evolution in production code to an AST element in test code.

the calls to *aMethod*. Evolution #2 depicts one of those calls located in test code, so that the analysis stops the exploration for this AST path: *it has found an evolution in production code that may have an impact on test code.* The analysis however continues to explore the AST based on the other calls to *aMethod* to find other potential impacts in test code.

5.2.3 Organizing evolutions

The static analysis can only find *potential* impacts on tests. We need a dynamic analysis to qualify those potential impacts with precision. To do so, we observe compilation and test execution results.

A naive analysis would require to evaluate $2^n \times m$ cases, where n is the total number of evolutions and m is the total number of tests: for each identified evolution, we would have to compile and run all the tests. For a large number of evolutions and tests in real histories this would not scale. Moreover, all evaluated cases might not be useful to find co-evolutions. Our third step $\boxed{C}$ thus aims at reducing this combinatorial number of cases. We developed an optimization based on the dependency graph (computed in the previous step) and complex evolutions. This optimization consists in organizing atomic evolutions into groups that we are then able to explore efficiently (see Section 5.2.4).

Grouping statically related evolutions

The static analysis of the previous step computed a set of dependencies from production code to test code, as illustrated by Figure 5.5. Instead of considering each dependency path individually we can group those that are statically related. We consider evolutions not statically dependent on each other as unlikely to be part of a same co-evolution. The top graph of evolutions of Figure 5.6 shows an example of a statically related group of evolutions.

Grouping evolutions by source

Based on the groups of evolutions produced during the previous grouping step, the second grouping step simply consists in grouping those groups using their location in the production code or tests.

Grouping using complex evolutions

The approach handles atomic evolutions in the grouping step $\boxed{C}$ because atomic evolutions are: i/ the basic evolution operations so that they also represent complex evolutions; ii/ the most accurate evolutions to represent the transition from a state to another. For the detected complex evolutions, to be able to apply them and qualify their effects, we thus represent them as a group of atomic evolutions. So, a complex evolution is applied if all its atomic evolutions are applied.

For example, moving a method is a complex evolution composed of multiple atomic ones: it consists of deleting the method from its original location and inserting the same method in a different location, as depicted by the "*Graph of grouped evolutions*" in Figure 5.6: the deletion D_0 implies D_3 and D_4 so that they are grouped. The same reasoning applies for the insertions

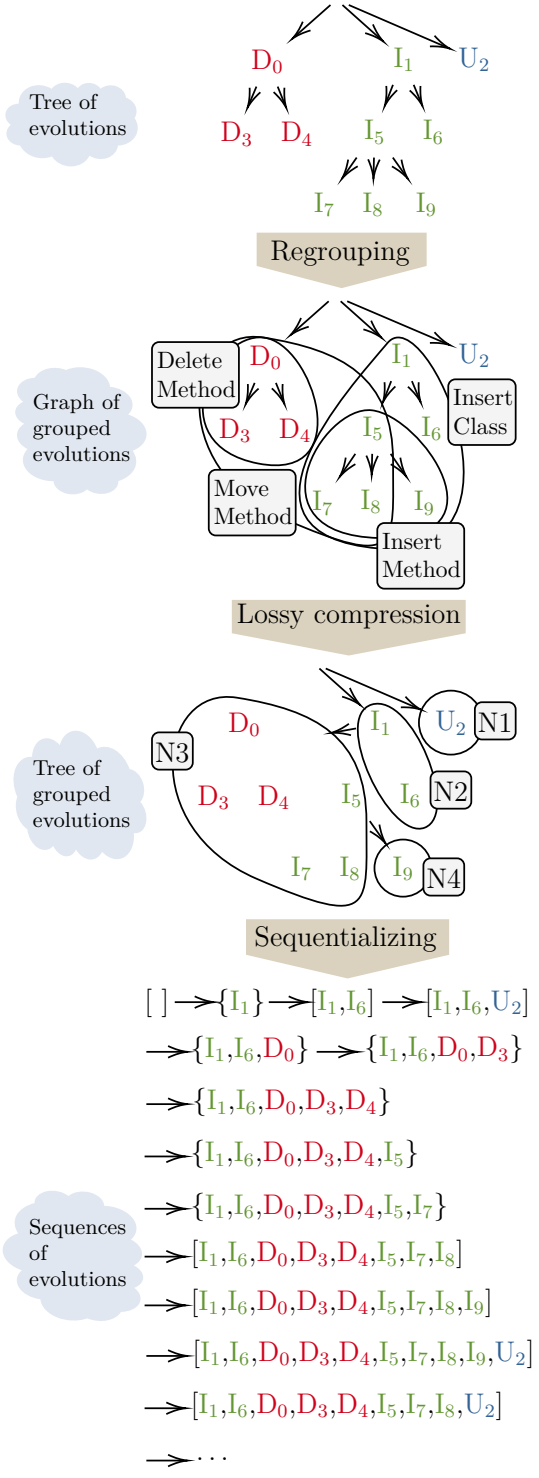


Figure 5.6 – The different steps for organizing and sequentializing the evolutions to be then qualified.
Legend:
D_x, I_x, U_x: AST element deletion, insertion, update (respectively)
[...]: sequence of evolutions to evaluate
{...}: intermediate sequence of evolutions (not to evaluate)

where: I_5 (resp. I_7, I_8) corresponds to its opposite operation D_0 (resp. D_3, D_4). The operation I_9 is not part of *Move Method* since it is a post-operation applied after the move to customize the insertion.

Lossy evolution graph compression

The previous grouping step produces as output an oriented graph of grouped evolutions that is still too large and not usable for an automated and efficient exploration. This new step transforms the graph of grouped evolutions into a *tree of evolutions* (see Figure 5.6). To do so, the transformation starts from the root node and analyzes its children, namely D_0, I_1, and U_2. The transformation then considers their upper group of evolutions and parenthood that links those groups:

— U_2 has no upper group so it is left alone ($N1$).

— The *Move Method* group requires the target class to exist so that the *Insert Class* group must be its parent (so $N2$ is the parent of $N3$).

— Three atomic evolutions of *Insert Class* (I_5, I_7, and I_8) are related to the *Move Method* group, so that the *Insert Class* can be reduced to I_1 and I_6 (group $N2$) and have as unique child the *Move Method* group (group $N3$).

— I_9 refers to an additional insertion done after the move, so that it is left alone in $N4$ and has as parent $N3$.

Compared to the initial graph of atomic evolutions, this step significantly reduces the number of cases to consider.

5.2.4 Qualifying evolutions effects

Sequentializing evolutions

In step $\boxed{D}$ (Figure 5.5), to evaluate the functional effects of each group of evolutions on tests, we need to apply those evolutions to then run the tests. To do so, our approach now takes as input the tree of groups of evolutions produced by the step '*Lossy compression*' (Section 5.2.3). The approach now *sequentializes* the compressed tree (see Figure 5.6), *i.e.,* it transforms this tree into sequences of evolutions to apply. This transformation uses a gray code [1] to go through all possible combinations of leafs in the tree of evolutions. Then, parents evolutions are interleaved, following two cases: if the leaf corresponds to an insertion then its parents should be inserted recursively (if not already done); if the leaf corresponds to a deletion then its parents with no child must be removed. Note that the entire group of evolutions must be applied before we can run the tests.

For example, with Section 5.2.3, at the root node the sequence is empty. The first child is I_1, but this sequence is incomplete (I_6 missing) so that it cannot be evaluated. I_6 is then added

[1] An optimal ordering algorithm for sequences of evolutions. See "Section 22.3. Gray Codes" in Press *et al.* [Pre+07].

to complete this sequence. Recursively, atomic evolutions of $N3$ complete another sequence to evaluate: $[I_1, I_6, D_0, D_3, D_4, I_5, I_7, I_8]$. I_9 then completes another sequence.

Qualifying sequences of evolutions

The qualification is based on compilation and test execution results. For example with Figure 5.6, we assume that those changes compose the commit t_x. The approach applies on the code at t_{x-1} (*i.e.,* before applying t_x) the first possible sequence: $[I_1, I_6]$. This sequence gives as output one of the qualifications we introduced previously: PCFAIL, TCFAIL, TFAIL, or TPASS.

It then applies the next sequence: $[I_1, I_6, U_2]$. In an optimization purpose, the approach does not go back to t_{x-1} and apply the full sequence. Instead, it applied on $t_{x-1} + [I_1, I_6]$ the missing operations, namely U_2.

When the next sequence to apply does not contain operations applied during the previous sequence, the approach revert those operations. For example, with the last two sequences of Figure 5.6: $[\ldots, I_8, I_9, U_2]$ and $[\ldots, I_8, U_2]$. The approach must revert the operation I_9 to obtain the latest sequence. Because our approach relies on the three basic evolution operations (insert, delete, update), to cancel one of these operations the approach applies its opposite: for an insert it applies a delete (vice versa); the opposite of an update is an update that automatically applies its changes in the reverse order.

5.2.5 Assembling co-evolutions

Based on the effects of each group of evolutions, this step $\boxed{E}$ (see Figure 5.6) can now combine evolutions to form co-evolutions. For example, given a group of evolutions A that breaks a test T and an additional group of test evolutions B that fixes back T. Our approach then identifies a co-evolution where A is the cause of the co-evolution and where B is the repair of the co-evolution. Based on definitions 6.B and 6.A, we can qualify the types of co-evolution as *complete* or *partial*.

To find delayed co-evolutions (def. 4.D), spanning over multiple commits, we search for a group of evolutions that breaks a test in some commits, then we search the same test (using the signature along with move and update evolutions) in a later commit where we locate groups of evolutions that repair the test. It should be noted that, here, evolutions from intermediary commits are either considered as part of the cause, or the repair (depending on status of the test before applying the final repair), whether all intermediary evolutions are part of the actual minimal co-evolution would require additional refinements and is left as a future investigation.

5.2.6 Implementation

We implemented our approach to support Java Maven projects that use Git. Our implementation is open-source and freely available in our companion Web page.[2]

[2] `https://github.com/quentinLeDilavrec/ICSME2021`

Technically, it is implemented in Java and interfaces with *Jgit*, *GumTree* [Fal+14], *RefactoringMiner* [Tsa+18a; TKD20], and *Spoon* [Paw+15] for performing the steps $\boxed{A}$ to $\boxed{C}$ (see Figure 5.4). The tool stores all the results in a *Neo4j* graph database.

For qualifying groups of evolutions (step $\boxed{D}$) the tool relies on Maven and its build lifecycle phases that permits to: compile production code, compile test code, run specific tests, and get the status of these phases. For example, to qualify complete co-evolutions, Maven must gives us:

1. TPASS before applying the *cause* evolutions;
2. TFAIL or TCFAIL right after those evolutions;
3. TPASS again after having applied the *repair* evolutions.

One can query all the results in the database using the Cypher query language [Fra+18]. We did so to extracting data we discuss in the next section dedicated to the evaluation of the proposal based on this implementation.

5.3 Evaluation

This section presents the evaluation of our approach. All the materials and data of this empirical study are freely available on the companion Web page. First, we present the research questions. Then, we present the data set and evaluation process before we discuss the obtained results. We finally discuss the threats to validity and the scope of the approach.

We ran the implementation of our approach on the following hardware configuration: *2 x Intel(R) Xeon(R) Gold 6238 CPU @ 2.10GHz; 187Gb ram; 1 T SSD*, running *Ubuntu*.

5.3.1 Research Questions

RQ1: Can we detect code and test co-evolutions with precision? This aims to assess the ability of our approach in finding co-evolutions, by looking both at their quality and representativity.

RQ2: Can we detect immediate and delayed co-evolutions and in what ratio? This aims to assess whether our approach can find both types of co-evolutions and to shed light on their frequency.

RQ3: To what extent delayed and immediate co-evolutions are similar or different? Through this question we aim at scrutinizing those co-evolutions to characterize them.

RQ4: What is the time performance of the approach? We aim at discussing to what extent the approach scales.

5.3.2 Experimental Protocol

To the best of our knowledge, our approach is the first one that permits to automatically find code and test co-evolutions from histories. So, we had to select representative software systems to analyze. This sub-section details the selection criteria we used and the resulting data set of software systems. We then detail the dependent variables.

Data Set

The source of our data set is GitHub. We aimed to select Java projects that we can compile, are well tested and having a rich evolution history. To gather such data, we took the following steps:

1. We queried GitHub [3] to retrieve Java projects with more than five stars and that use Maven (*i.e.,* have a *pom.xml* file) so that dependencies, builds, and tests are automatically handled. We then made an intersection with the qualitative data set [4] of Allamanis *et al.* [AS13] who aimed to study coding and testing practices in Java. With this first step, we found 3588 repositories that should now be filtered.
2. We then selected projects that have at least two releases on GitHub. This selection has multiple beneficial effects: (a) We consider that as a possible maturity threshold; (b) Zaidman *et al.* [Zai+11] have shown clear development cycles between releases; (c) It filters out initial commits, which are often difficult to interpret due to their size and uniqueness; (d) We made the hypothesis that the longest a code history is, the more it might contain test and co-evolutions (because of maintenance and evolution operations); This filtering resulted in 3588 repositories.
3. Even though we selected projects with releases, it is still not a guarantee that they compile and build. Thus, for each project we tried to construct the AST and build the ten latest releases and filtered those that did not build at least once. Thus, we kept at the end 395 repositories with 5439 releases in total (with an average of 14 releases per repository).
4. This work focuses on code and test co-evolution. Tests are also crucial in our approach since it uses tests to qualify the nature of the effect of evolutions. So, we further selected projects that have a significant number of tests (JUnit tests in our case). We fixed this minimal threshold of tests to 50. This resulted in 164 repositories to analyze.
5. As we consider complex and refactoring evolutions in this chapter, we aimed at detecting co-evolutions involving refactorings. As such, we found 30 122 refactorings on 91 projects. Then, we extracted the ones with at least 10 refactorings, thus, **reaching 45 repositories**.

Dependent Variables

Precision. For measuring the precision, we first applied our automated approach to obtain a set of possible co-evolutions. We then check them manually, but since this is a very time-consuming task, we only scrutinized *complete* co-evolutions to state whether they were indeed valid ones. On these *complete* co-evolutions, we computed the precision as follows.

$$precision = \frac{|coevolutions_{correct}|}{|coevolutions_{detected}|} \times 100$$

Recall. For measuring the recall, we need to have a ground truth of test and code co-evolutions. Such a ground truth would permit the identification of the co-evolutions our approach

[3] Using *PyGithub.py* library `https://github.com/PyGithub/PyGithub`

[4] `http://groups.inf.ed.ac.uk/cup/javaGithub/`

does not detect. However, to the best of our knowledge our approach is the first one that automatically detects code and test co-evolutions, so that no ground truth exists.

Manually detecting co-evolutions in the analyzed repositories to obtain a ground truth is not possible: for immediate co-evolutions, this would require to manually test the effects of several evolutions together. So for a commit with n atomic evolutions, this would imply $2^n \times m$ cases to evaluate. It is even more challenging to start looking for delayed co-evolutions as the number of atomic evolutions to consider mechanically increases. The evaluation of one single case requires to run each test before and after the code evolutions, and after the tests evolutions to qualify the evolutions as co-evolutions or not. This becomes infeasible to perform manually.

Nonetheless, we can compute the minimal recall to have an underestimation of it. To do so, when a code evolution statically impacts a test, we make the overestimating hypothesis this test is part of a code and test co-evolution. So, given $|tests_{impacted}|$ the number of tests we found statically impacted by evolutions, and $|tests_{coevolved}|$ the number of such tests already part of the co-evolutions we identified, we computed the recall this way:

$$recall = \frac{|tests_{coevolved}|}{|tests_{impacted}|} \times 100$$

Execution time. Through the execution time we aim at evaluating the scalability of the proposed approach (RQ4). This variable computes the mean execution time of the approach on: one commit; one release; one repository. Given tt the overall execution time in seconds spent for analyzing all the repositories, $|repos|$ the total number of analyzed repositories, $|commits|$ the total number of commits computed from all the repositories, we compute $time_p$ (time per repository) and $time_c$ (time per commit) as follows:

$$time_p = \frac{tt}{|repos|} \qquad time_c = \frac{tt}{|commits|}$$

5.3.3 Results

From the 45 repositories, we analyzed in total 6738 commits and 78 million lines of Java code.

RQ1: Can we find prescise code and test co-evolutions?

Table 5.1 reports the co-evolutions found in the analyzed repositories.

Regarding complete co-evolutions, in total, our approach automatically found 202 co-evolutions among which 88 co-evolutions involved at least one refactoring evolution (first line of Table 5.1). These values concern both immediate and delayed co-evolutions (we discuss in details about these two different types of co-evolutions in RQ2). We computed a precision of 100 % and a recall of 37.5 % for complete co-evolutions. To compute the recall, the value $|tests_{impacted}|$ equals 4997, while the value $|tests_{coevolved}|$ equals 1877. As previously discussed, our recall result is underestimated. To precise this value, we encourage researchers to replicate our experiment to potentially find missing co-evolutions. The co-evolutions we identified would serve them as the minimal ground truth for computing their recall.

Regarding partial co-evolutions, table 5.1 reports the results for all found co-evolutions in the different cases of the initial states of the co-evolved tests. We found 111 additional partial co-evolutions where the tests were failing, then impacted by the code evolution before to be repaired to their initial state. We also observed 276 passing tests and 23 failing tests where they respectively were degraded but did not come back to their initial state after being partially repaired.

Table 5.1 – Complete and partial co-evolutions found.

Status of tests		**Co-evolutions**		**Type**
Initial	**Final**	**All**	**With ≥1 refactoring**	
TPASS	TPASS	202	86	Complete
TFAIL	TFAIL	111	30	Partial
TPASS	TFAIL	79	13	Partial
TPASS	TCFAIL	197	115	Partial
TFAIL	TCFAIL	23	1	Partial
	Total	612	245	

Regarding the semantic of partial co-evolutions, we identified two different facets. First, a partial co-evolution can be part of a complete co-evolution (our implementation has found or not), *i.e.,* a step towards completing a co-evolution (we discuss the co-evolutions our tool may miss in the threats to validity section). Such partial co-evolutions are interesting to understand how complete co-evolutions are formed. For example, partial co-evolutions can be used as a contrast to complete co-evolutions for drawing lists of best and bad practices in code and test co-evolutions, analogously to design pattern and anti-pattern [Jaa+16]. Second, a partial co-evolution may exist because some tests did not pass, *i.e.,* developers never fixed them. We think that such partial co-evolutions are in minority in our results because of the criteria we used to select mature projects.

RQ1 insights:

- ❖ Our approach founds complete code and test co-evolutions with a precision of 100 % and a recall of at least 37.5 %.
- ❖ We give evidences of the existence of *partial co-evolutions*. Finding partial co-evolution shows our approach flexibility and ability to cover different scenarios.
- ❖ Thanks to those co-evolutions, we produced a knowledge base that practitioners can use, for example, as a ground truth in experiments.

Table 5.2 – All detected categories of co-evolutions.

Status of tests		**Immediate Co-evolutions**		**Delayed Co-evolutions**		**Type**
Initial	**Final**	**All**	**With ≥1 refactoring**	**All**	**With ≥1 refactoring**	
TPASS	TPASS	140	82	62	6	Complete
TFAIL	TFAIL	62	28	49	2	Partial
TPASS	TFAIL	78	12	1	1	Partial
TPASS	TCFAIL	197	115	0	0	Partial
TFAIL	TCFAIL	23	1	0	0	Partial
	Total	500	238	112	3	

RQ2: Can we find immediate and delayed co-evolutions and in what ratio?

Table 5.2 displays our results for immediate and delayed co-evolutions for both complete and partial co-evolutions. Regardless of the type of co-evolutions (complete, partial, immediate, delayed), our approach found a total of 612 co-evolutions. The total number of immediate co-evolutions (complete and partial) is 500, so 81.69%. The total number of delayed co-evolutions (complete and partial) is 112, so 18.30%. Moreover, we observe a slightly different ratio on the 202 complete co-evolutions with a bit more of immediate ones. In particular, 140 (69.30%) are immediate complete co-evolutions and 62 (30.69%) are delayed complete co-evolutions.

Finally, we calculated the gap in delayed co-evolutions as the number of commits between the commit of the cause and the repair. For example, a gap of two means that the cause is in a first commit and the repair in the next commit.

We observed that 69, 3, 1, 13, and 26 of the delayed co-evolutions (both complete and partial), respectively, come with a gap of 2, 3, 4, 5, 8. Regarding complete delayed co-evolutions, 59 come with a gap of 2, while 3 others, respectively, come with a gap of 3, 4, and 5 commits. Note that in order to scale when detecting delayed co-evolutions, we limited our search up to a gap of 30 commits.

RQ2 insights:

- ❖ Our approach is able to automatically spot *immediate* and *delayed* co-evolutions.
- ❖ We observed a ratio of 69.31% of immediate complete co-evolutions and 30.69% of delayed complete co-evolutions, suggesting that developers seem to co-evolve their tests while performing evolutions on production code.
- ❖ We observed mostly a distance of two commits in delayed co-evolutions while the rest varied between four and eight commits.

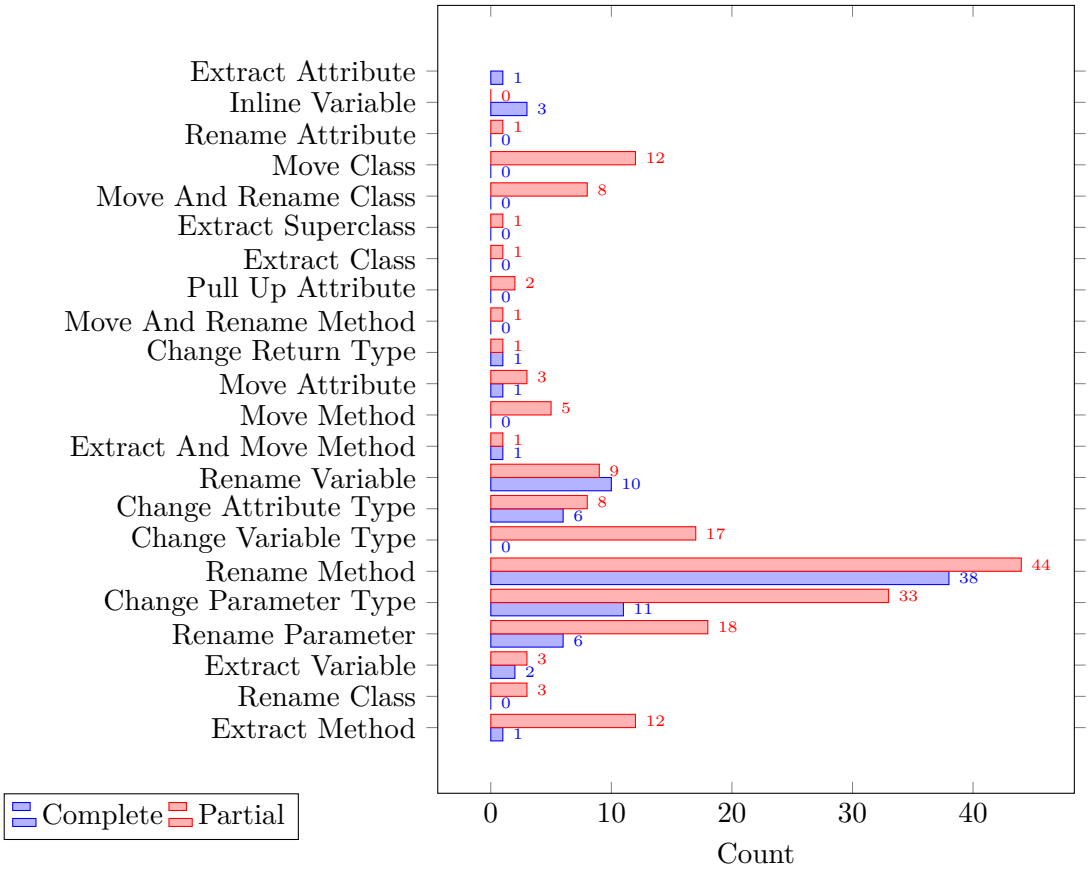


Figure 5.7 – Refactoring evolutions in immediate co-evolutions.

RQ3: To what extent delayed and immediate co-evolutions are similar or different?

We now look in depth at what are the differences or similarities of immediate and delayed co-evolutions.

First, we had a look at the evolutions constituting the detected co-evolutions, in particular at what refactoring evolutions are part of both types of co-evolutions. Figures 5.7 and 5.8 give the types of refactoring and their frequency we found in respectively the spotted 272 refactorings in immediate and 14 refactorings delayed co-evolutions.

We observed that when a refactoring evolution is part of a code and test co-evolution, it is most likely to occur within an immediate co-evolution (272), whereas, delayed co-evolutions rather implicate atomic changes and few refactorings (14). This suggests or can be explained as when developers perform a significant evolution to the code, they may tend to ensure their consistency with the tests immediately, hence, performing immediate co-evolutions. Whereas, developers may miss impacts of small or minor code evolutions on the tests, hence, performing delayed co-evolutions.

Moreover, we investigated the length of the dependency chains (in edges) in both types of co-evolutions. Our hypothesis is that the shorter the dependency chain is, the likelier the co-evolution to be immediate, and vice versa. On average, the length of the dependency chain for an immediate and a delayed co-evolution, respectively, is 10.84 and 10.44. Thus, our hypothesis

is rejected based on the results. This is surprising, because one may think as the more close the code evolution to the tests, the likelier developers will see it and fix it immediately and not delay it in further commits. Nonetheless, future research remains necessary to further investigate other reasons behind the delayed co-evolutions.

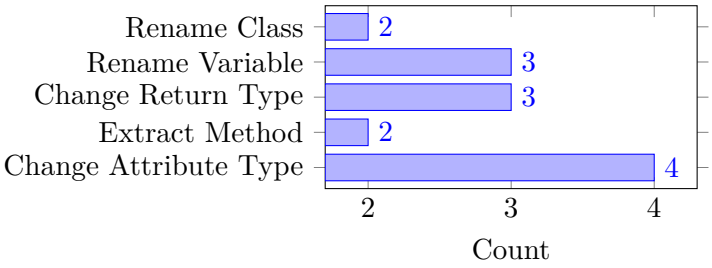


Figure 5.8 – Refactoring evolutions in delayed Co-evolutions.

RQ3 insights:

- ❖ Immediate co-evolutions tend to have far more refactoring evolutions (272) than in delayed co-evolutions (14). This may suggest how developers handle co-evolutions based on the size of the committed evolutions.
- ❖ Our hypothesis that the immediate co-evolutions tend to have shorter dependency paths between the cause and the repair than the delayed co-evolutions is rejected.

RQ4: What is the time performance of the approach?

The overall execution time (tt) our tool spent for analyzing the whole 164 ($|repos|$) repositories [5] is 90 720 min (1512 h). The value $|commits|$ is 15 954.

So $time_p$, the average time per repository, is 553.17 min (9.22 h). $time_c$, the average time per commit, is 5.67 min. Of course these mean values depend on: the number of commits per repository; the number of evolutions per commit. As an indication, Figure 5.9 depicts a boxplot of the number of evolutions per commit, with a median value of 10 atomic evolutions per commit.

RQ4 insights:

- ❖ The computed durations show that our proposal scales but still requires considerable resources to perform large-scale studies.
- ❖ The time it takes to automatically analyze the code base typifies the extreme difficulty to perform such analyzing tasks manually, thus motivating our proposal.

5.3.4 Threats to Validity

Beyond the threats to validity that may affect our experiment, this section also discusses the scope of our proposal.

[5] We had to apply the approach on the 164 to perform the filtering step 5) on the data set. We consider here those 164 repositories instead of the curated set of 45 repositories to have more representative results.

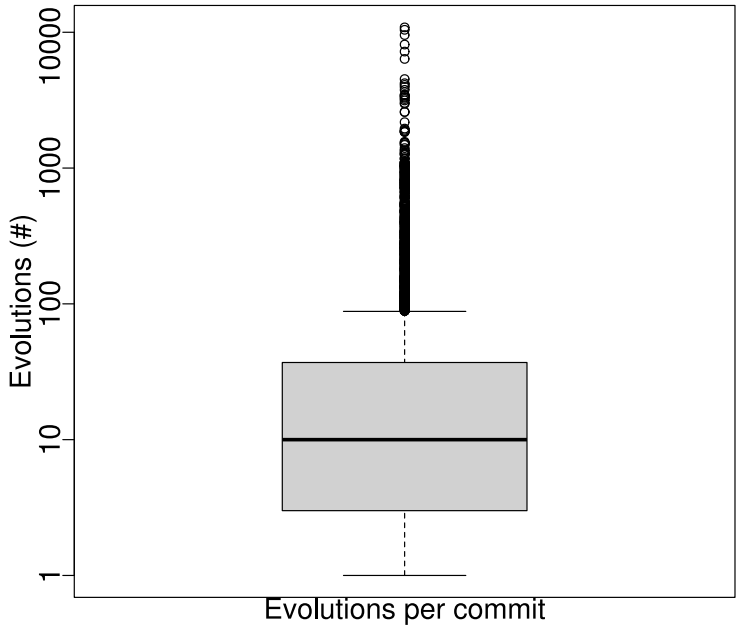


Figure 5.9 – Number of atomic evolutions per commit

External Validity

We used selection criteria to find a large representative panel of Java code repositories, stored on Github. We consider Maven projects, a build automation tool widely used in the industry. We do not think that the use of another build automation tool, such as Gradle, would have an impact on detection of code and test co-evolutions.

We use Github releases during the selection process of projects to analyze. A Github release is associated to a Git tag. This helped us in filtering Git tags of interests. We do not think that the use of other version control systems (VCS), such as Mercurial would prevent the use of our proposal since it is based on atomic evolutions common to all VCS. The main challenge is related to RefactoringMiner that our approach relies on for detecting refactorings and that works with Git. We would have to find a similar tool for other VCS. Finally, we cannot generalize our results to software repositories in other languages than Java.

Internal Validity

We conducted manual analyzes to measure the precision of the approach. Such a manual analysis is error-prone, so to overcome this threat two persons performed all the manual analysis. They then compared their results. On divergent results, these two persons discussed to converge to a final decision. These persons are authors of the paper with a high expertise in detecting code and test co-evolutions. Note that as non-experts of the analyzed projects, these two persons cannot detect tests of poor quality, *i.e.,* tests that still pass while they should not (missing assertion, flaky tests, etc.). To limit this issue, we designed criteria for selecting relevant projects.

The exact computation of the recall is not technically possible due to the combinatorial explosion of the number of cases to evaluate when establishing ground truth. To overcome this issue we underestimated the real recall of our approach by over-estimating the possible existing co-evolutions. This way, we do not overclaim on the performance of our approach.

Construct Validity

Our approach relies on *Gumtree* and *RefactoringMiner* to find atomic and complex evolutions. The precision and recall of these tools have an impact on the performance of our proposal. To limit this threat we validated by hand the co-evolutions found by our approach.

To qualify evolutions, we rely on code compilation and test execution verdicts. However, a passing test suite after a production code change does not certify that this change has no effect on the test suite: a test of low quality might not spot regressions. We consider this threat while designing the selection criteria of the projects to analyze. Moreover, one can consider other assessable properties that compilation result and test verdict to spotting co-evolutions. We think that compilation results and test verdicts are still good enough to cover a large majority of test and code co-evolutions.

Closely related, the relevance of the selected projects has an impact on the evaluation results. To limit this threat, we designed rigorous selection criteria.

Regarding the recall of our approach, we use our static analyzing technique for spotting tests that are statically impacted by evolutions. Our technique might miss some cases, in particular because our approach does not consider code reflexivity.

Regarding the detection of delayed co-evolutions, we use a maximal threshold of 30 commits between commits to analyze because of resource limitations. If such a threshold may miss complete co-evolutions we think that its value 30 is high enough to limit this effect.

Finally, Git and Github have several strategies for merging branches that may affect the computation of immediate and delayed co-evolutions. For example, the *Squash* technique squashes all the commit of the branch into a single commit. The *Rebase* technique individually adds each commit of the branch into the base branch. Studying commits and branches structures is another combinatorial challenge. We only considered the most direct sequences of commits between releases.

5.4 Conclusion

To the best of our knowledge, the approach presented in this chapter was the first to automatically find code and test co-evolutions in object-oriented code. Our approach can find co-evolutions contained in a single commit (immediate co-evolution) but also co-evolutions scattered over multiple commits (delayed co-evolution). We implemented our approach in a tool that analyzes Java code stored in Git repositories. Our implementation is fully open-source and available for replicating studies.

We evaluated our approach by conducting an empirical study on a curated data set of 45 repositories using our implementation. Our approach found 202 complete and 410 partial co-evolutions. We also detected 140 immediate and 62 delayed complete co-evolutions. For delayed co-evolutions, the gap between the cause and repair commits varied from 2 to 8. Finally, we also observed that the dependency chains tend to be the same in immediate and in delayed co-evolutions.

An internship future work would be to explore further composition of tests, together with

more aggressive delta refinements on specific co-evolutions, especially the delayed ones to try to observe kind of trickle co-evolutions. Initially, we envisioned the use of this ground truth to design a recommendation engine for code and test co-evolution. We also aimed at porting over the ground truth to analyze, understand, and explain at a large scale how developers do test and code co-evolutions. We also planed to consider other software artifacts in addition to code and test, such as issues and dependency configurations. In particular, considering dependencies may increase the quantity of found co-evolution.

But there were obvious performance limitations that were observed during the evaluation, that will be discussed with more details in the implementation section 9.1.1. To sum up, we had a suboptimal memory management due Java and a poor synergy between the implementation of Spoon, Gumtree, and RefactoringMiner, itself in part due to limitations of Java at composing objects other than by reference, becoming very inefficient for large numbers of such compositions *e.g.,* elements of ASTs (both for garbage collection and cache locality). While on a more fundamental standpoint, we would have benefited from an approach that would have enabled us to leverage temporal and spatial redundancy in source code histories.

To address these limitations, we proposed an approach, called *HyperAST* (chapter 6), that leverages the temporal and spatial redundancy in source code histories at the granularity of a CST and with features of an AST. To show its benefits and in relation to our approach detecting co-evolutions we focussed on demonstrating two components: Chapter 7 proposes a reference analysis which is incremental and targets impact analysis; Chapter 8 proposes a structured diffing (inspired by *GumTree*) that supports efficient tracking of code elements.

Chapter 6

HyperAST: efficient analysis of entire code histories

The contributions presented in this chapter directly target the resolutions of limitations presented in the previous chapter, with the idea of reusing non changed parts of AST through time.

To recontextualize, the emergence of distributed version control systems (VCS), such as GitHub or GitLab, has permitted accessing a massive quantity of software and their histories. This offers golden opportunities for both researchers and engineers to perform code analysis of software at large.

Researchers have been analyzing software code histories from different angles, such as recovery of traceability links [AHS20; Bav+12], refactorings [TKD20], edit scripts (aka. Diffs) [Fal+14], duplicate code [Maz+16], bad smells and their origins [Tuf+15], or mining of fixes for program repair [Koy+20]. Engineers in the software industry developed code analysis tools at scale, such as Copilot [1] for code completion, LGTM [2] for bug detection, or CodeQL [3] for querying the code.

To do so, relying on structured code representation, namely the Abstract Syntax Trees (AST), has become a foundation for many of the above-mentioned tools and software engineering activities.

One of the intrinsic property of software is its continuous evolution [Men08] and its growing complexity as it evolves. This new dimension asks for temporal code analysis of software histories. In particular, to consistently analyze code elements through their evolutions (*i.e.,* different commits and releases) and also linking their analysis. Such a temporal code analysis can focus, for instance, on origin of code smells [Tuf+17], bug prediction [Pal+17], class stability [Rap+04] throughout commits, co-evolution [Le +21; LY17] by linking impacting changes and their resolutions.

However, a temporal code analysis requires to simultaneously handle multiple ASTs, corresponding to the different versions of the corresponding software across its history. Unfortunately, doing so on large set of commits for large sized software faces major scalability issues both in terms of memory and CPU usage. With state-of-the-art analysis tools, the whole computation is indeed typically redone from scratch for each version, *i.e.,* commit, even if the commit under analysis is almost identical to previously analyzed ones [LY17; Tuf+17; Le +21]. Boldi *et al.* [Bol+20] proposed to compress the file structure of a software history without considering the

[1] https://copilot.github.com/
[2] https://lgtm.com/
[3] https://codeql.github.com/

content and for storage purposes of the archive. With LISA [Ale+19], Alexandru *et al.* rely on a vertex compression algorithm to share AST nodes among revisions. However, without sharing identical code subtrees independently of their position, nor providing tracking of changed code elements.

The contribution presented in this chapter is to add a time dimension to an AST, turning it into a *HyperAST* capturing all of it history. The goal is to ease temporal code analysis at large scale on a large timeline of a software history. To do so, the *HyperAST* is built incrementally on a set of commits. It integrates the ASTs of the different versions in one place by leveraging on the code redundancy through space (between code elements) and through time (between versions). This stems from the observation that for a set of commits in a software history, most of the ASTs' elements are similar since most often only small code changes are applied in each commit compared to the rest of the code base. In its core, the *HyperAST* is a Direct Acyclic Graph (DAG) where nodes are unique, allowing for an efficient reuse of nodes in a single version and across versions if unchanged. Moreover, intermediate computation results on top of the *HyperAST*, such as hashes and references, are calculated and stored as metadata along the nodes of the *HyperAST*. This allows for an efficient reuse of the metadata across versions. The goal of metadata is also to serve as basis for a further in-depth temporal code analysis. To the best of our knowledge, the *HyperAST* is the first attempt to offer an optimized AST covering multiple versions at once in contrast to maintaining multiple ASTs for each version.

We evaluate the *HyperAST* on three levels: 1. its feasibility, 2. its scalability, 3. its usefulness on a practical scenario of usage, namely finding references of declarations and computing tree diffs (presented in chapters 7 and 8).

In particular, we evaluate the *HyperAST* on a data set of 18 real-word and representative Java projects taken from GitHub, by comparing it on a sample of thousand of commits per project to a state-of-the-art tool as a baseline, namely Spoon [Paw+15]. Our results show the *HyperAST* correctly represents a software history of a set of commits. It nearly always was able to serialize the code back to its files. Only in 0.02% of the cases it failed, due to erroneous code in its original state. Moreover, we were able to scale on large repositories with an order-of-magnitude difference in speed between the *HyperAST* and Spoon, from 6 up to 8076 in CPU construction time and from 12 up to 1159 in memory footprint. The minimum and a maximum of construction time in CPU for the *HyperAST* ranged from 1 min and 2 h 22 min, while it ranged from 1 h 14 min and 93 h 31 min for Spoon. Besides, The minimum and a maximum of memory footprint for the *HyperAST* ranged from 63 MB and 7.2 GB, while it ranged from 16 GB and 2.2 TB for Spoon. Thus, the gains in construction time varied from 83.4 % to 99.9 % and the gains in memory footprint varied from 91.8 % to 99.9 %.

The main contributions presented in this chapter are:

1. a novel kind of AST, namely the *HyperAST*, that aims to enable efficient large scale temporal code analyzes on software histories;
2. an open-source implementation of the *HyperAST*;
3. an evaluation that demonstrates the feasibility, the scalability and the relevance of the *HyperAST*;

4. a replication package for open-science.

The rest of the chapter is structured as follows. Section 6.1 presents the *HyperAST*. Section 7.2 details the evaluation. Section 6.3 concludes the chapter.

The approach and results presented this chapter were published at ASE 2022 [Le +22].

6.1 The *HyperAST* Approach

This section details the concept of *HyperAST*, that improves scalability when analyzing large software histories. The first enabling hypothesis is about 1) redundancy in space between code elements in a single version, and especially 2) redundancy in time among consecutive commits, where small changes are applied compared to the overall size of the code base [Zai+08]. The second hypothesis is that various computations can be done once on code subsets [DR14; Fal+14] then reused multiple times in space and time.

This section first gives an overview of the *HyperAST* and then describes its structure. After that, it details how the *HyperAST* is constructed and updated with every new commit. Later in the thesis, chapters 7 and 8 present two foundational analyses that leverage the *HyperAST*.

6.1.1 Overview

Figure 6.1 shows the overall workflow of the approach of building the *HyperAST* incrementally on new commits. Given the Git the Merkle DAG of a software history, per commit (green circle), the construction process explores the hierarchy of directories through the Git Trees (blue triangles), and parses the code files represented by Git Blobs (red squares). We rely on Tree-Sitter [Mic22] that parses the code into a CST (*Concrete Syntax Tree*). For each git object treated, the construction process associates its Object Identifier (OId) to its corresponding node in the *HyperAST* and stores it into a cache. Thus, if encountered again in following commits, those nodes are not inserted and simply reused across versions. Moreover, when processing an element to insert in the *HyperAST*, the process first extracts identifying data and checks whether the element is already inserted. If not, metadata are computed and inserted as well.

6.1.2 *HyperAST* Structure

This section introduces the structure of the *HyperAST*.

The *HyperAST* represents a Direct Acyclic Graph (DAG) *i.e.,* a recursive data structure represented by list of nodes, such that:

$$HyperAST = \{v \in V | v = (type, label, children, metadata)\}$$

type: refers to a rule name of the language grammar, such as class, method declaration, assignment.

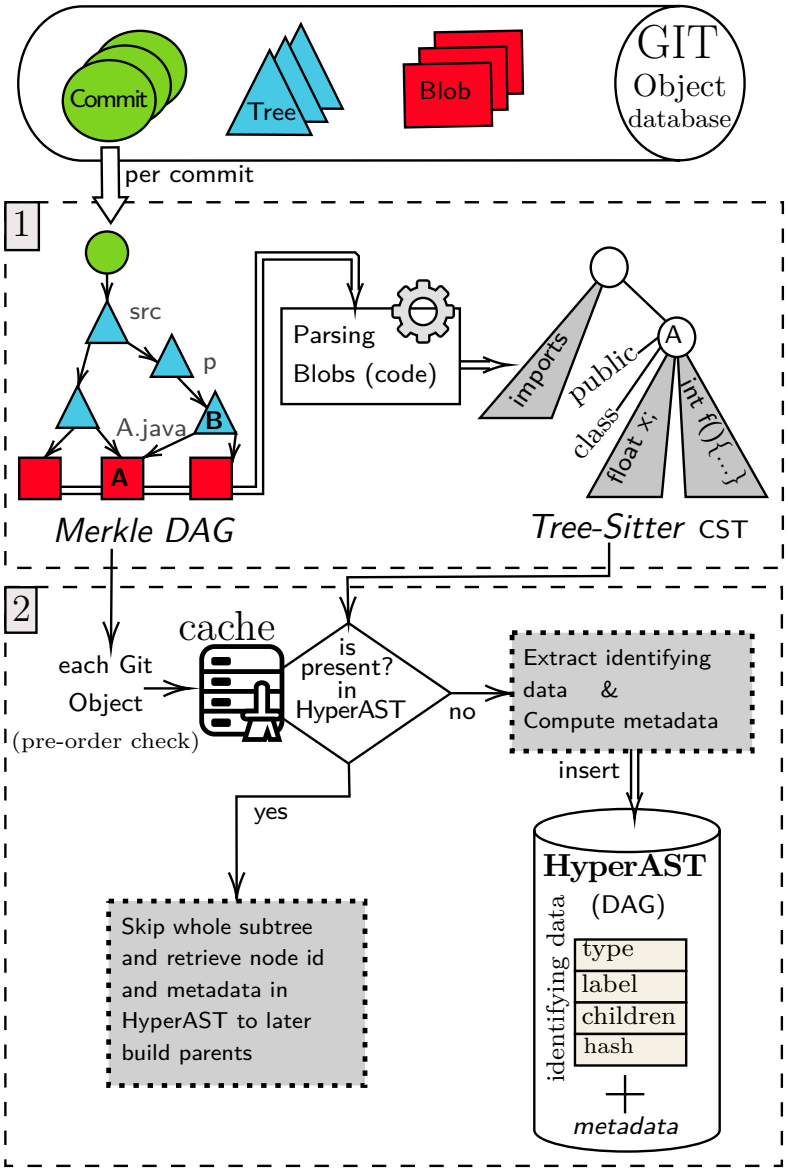


Figure 6.1 – Overall approach of the *HyperAST*.

label: The textual content of a token in a grammar. It is mandatory for leaf nodes but optional for the parent nodes. For example with the given variable declaration "*int* $i = 0;$", i is the label of the variable identifier and 0 is the label corresponding to the literal assigned to i. The parent node's label of the variable declaration is empty.

children: A list of *references* to other nodes in the *HyperAST*. They can be intrinsic identifiers (*e.g.,* unique content hash) or extrinsic identifiers (*e.g.,* pointer, index).

metadata: A persistent storage for intermediate results or computation of code analysis on the local subtree throughout versions. It helps in reducing redundant and unnecessary computation when analyzing code histories. Its exact definition depends on how stakeholders plan to use and augment the *HyperAST* to fit their needs. For example, it could be complexity metrics or hashes, it could be stored directly inline or somewhere else in the heap (see implementation chapter 9.1 for details).

```java
public class A {
    public void a1() {
        B b = new B();
        this.a2(b);
    }
    public void a2(B b) {
        b.b1();
    }
}
```

(a) Version v_x: Method $a1$ creates a B object and calls method $a2$ with it.

```java
public class A {
    public void a1(int value){
        B b = new B(value);
        this.a2(b);
    }
    public void a2(B b) {
        b.b1();
    }
}
```

(b) Version v_{x+1}: Method $a1$ now has a parameter used to create b.

Figure 6.2 – Illustrative example of 2 versions of a Java class.

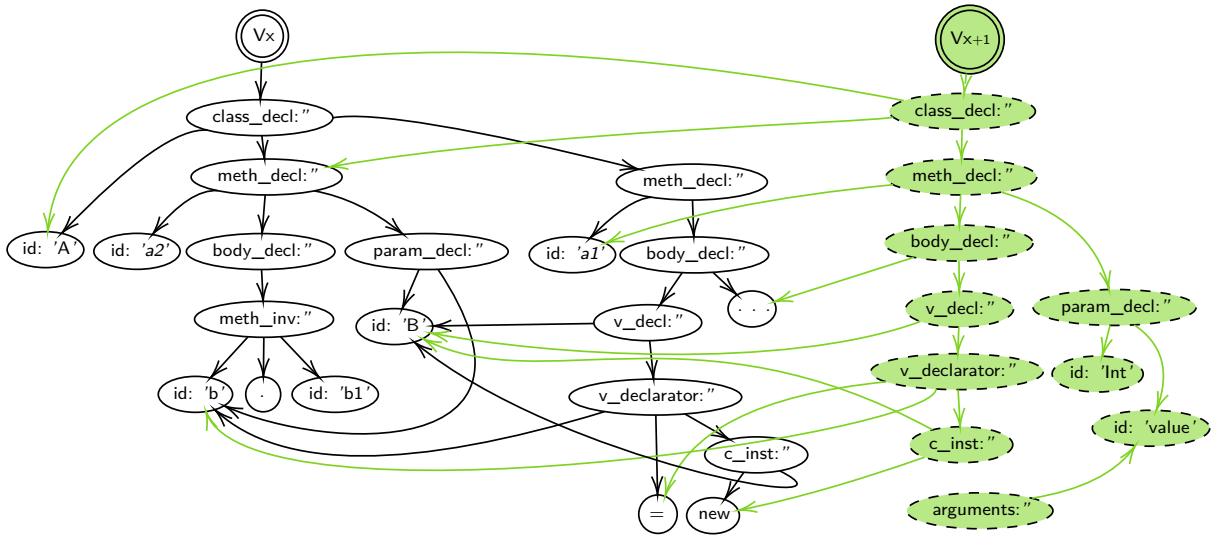


Figure 6.3 – Example of a representation in the *HyperAST* for the first and second commits of class A in Figure 6.2.

In the *HyperAST*, a node v is uniquely identified by its *type*, *label*, and *children*. This characteristic is essential when combined with the recursive structure of *children* as it allows reusing structural clones (*i.e.,* unchanged part of the code) from a commit to other ones.

6.1.3 *HyperAST* Construction

This section details the construction of a *HyperAST* by first focusing on the tree construction, and then on metadata computations.

Tree construction

The construction process of a *HyperAST* is a tree-to-tree transformation. It starts by processing the commits and their trees of a git's Merkle DAG recursively. This thus maintains the structure and hierarchy of the history in terms of directories and code files, facilitating the serialization of the *HyperAST* back to the code. When reaching a Blob, it is parsed as a CST to be processed before to be inserted into the *HyperAST*. So, the produced *HyperAST* keeps the same structure as the original git's Merkle DAG where Blobs are replaced by acpCST. Compared to Git, this increase of granularity enables code analysis and allows further code deduplication.

The construction of the *HyperAST* can be considered as append-only. Thus, it is incremental in its nature since the approach processes each commit separately and appends all elements related to the commit in the *HyperAST*.

Moreover, node insertion and metadata computation are done only if the node is absent from the *HyperAST*. To check whether an element is already present in the *HyperAST*, the approach uses its hash (see Similarity metrics in Section 6.1.3) to detect a structurally similar node. In the following, we explain how to handle the different Git objects and the parsed CSTs to construct the *HyperAST*.

Handling a Commit: Each commit has a corresponding root node added to the *HyperAST*.

Handling a Tree: To insert a Tree in the *HyperAST*, the approach first inserts its children (*i.e.,* post-order mode). To computationally benefit from the Merkle DAG, the *HyperAST* keeps a temporary association table as a cache between an Oid (git Object Identifier) and references to *HyperAST* nodes. This allows quickly checking the presence of a node in the *HyperAST*. The construction approach handles each child of a given Tree as follows:

Tree Child is handled recursively.

Blob Child is parsed to produce a CST, which elements are handled recursively.

Finally, a Tree is inserted in the *HyperAST* as a node with its type set to *'directory'* and its label as the Tree name, and with references to its processed children.

Handling a CST element: A CST is also a recursive structure, which is processed following the same steps as for handling a Tree in post order. Finally, a processed CST element is inserted in the *HyperAST* as a node with its: original CST element type; label; and references to its processed children.

We now take the two commits in Figure 6.2 as an example to show the constructed *HyperAST*. Figure 6.3 depicts the constructed *HyperAST* at the second commit $Vx+1$. The left part (white) depicts the *HyperAST* after its construction on the first commit Vx. The right part (green) depicts the new elements added to the *HyperAST* after the second commit is treated: the addition of the parameter `int value` in the method `a1()`, and the argument `value` in the constructor invocation `new B()`. This example shows how both commits exist in the same *HyperAST*. For sake of readability, Figure 6.3 does not illustrate the nodes' metadata. The entry points in the *HyperAST* corresponds to the handled commits (V_x and V_{x+1}). Thanks to the structural similarities between V_x and V_{x+1}, V_{x+1} **can reuse all unchanged subtrees of the code and their already computed metadata**, depicted by the green arrows: this is reuse of **redundancy in time**. Finally, it is worth noting that **nodes are shared even in a single version, *i.e.,* redundancy in space**, such as the `id: 'value'` node in dashed green.

Metadata provisioning

The goal of metadata is to provide developers with intermediate pre-calculated results to be reused for a posteriori temporal code analysis across versions. We designed the *HyperAST* to be extensible to add other types of metadata either during or after its construction by appending

the newly computed metadata. This section discusses two concrete kinds of metadata that we are computed during the evaluation.

Similarity metrics. During the construction of a *HyperAST*, hashes [Fal+14; CDR09; CAM02] are computed as metadata to help in comparing subtrees: a structural hash; a structural and label hash; a syntactical hash. The structural hash only uses the type of nodes, while the structural and label hash also uses nodes' labels. The creation of a *HyperAST* also computes for each subtree a syntactical hash that considers its serialized code. This metric aims to help in comparing and indexing versions of code. It is used in chapter 8.

Index of references. Given a declaration, finding all its references is a standard feature in most code analysis tools. It often implies maintaining an association table between declarations and references. This works fine in the case of a single version. However, maintaining such global tables and ASTs for multiple versions requires heavy maintenance work to keep those numerous tables synchronized. Instead, one could use an oracle capable of answering with certainty when a reference is absent from a given piece of code, *i.e.,* from a subtree. Thus, only exploring the subtrees that may contain references. It is used in chapter 7.

6.2 Evaluation

This section presents the evaluation of the *HyperAST* construction. First, we present the research questions to then discuss the results. Then, we present the data set and evaluation process. We finally discuss the threats to validity, limitations, and the scope of the approach. **All the material of this section and a replication package are available on our companion web page** [4].

We ran the implementation of our approach on the following hardware configuration: *2 x Intel(R) Xeon(R) Gold 6238 CPU @ 2.10GHz; 187Gb ram; 1 T SSD*, running *Ubuntu 18.04.6*.

6.2.1 Research Questions

We now formulate the research questions as follow:

RQ1 Can we compute the *HyperAST* correctly over several versions? This research question aims to investigate the sound construction of the *HyperAST*.

RQ2 How does the *HyperAST* perform and scale compared to a traditional approach? This aims to position the scalability performance of our contribution over a long evolution history with an established state-of-the-art solution.

6.2.2 Data Set

This section presents the data set used in the evaluation and its selection process. The source of our data set is GitHub. We aimed to select real-world Java projects that compile, that are widely used, and continuously maintained with a rich evolution history. To gather such data, we follow this process:

[4] `https://github.com/quentinLeDilavrec/ASE2022`

Table 6.1 – Data set characteristics.

Projects	Java LoC	Java files	Commits	Contrib.	Stars
Apache Hadoop	1.63M	10.2k	25,749	435	12k
Apache Flink	1.5M	13.2k	30,587	1,037	1.8k
Netty	317k	2.78k	10,789	569	29k
AWS SDK Java v2	265k	3.15k	8,766	88	1.4k
Apache Dubbo	197k	2.81k	5,437	393	37k
Apache Log4j2	183k	2.32k	12,031	132	2.8k
Jenkins	181k	1.69k	32,252	701	19k
Javaparser	179k	1.67k	8,031	166	4.1k
Inria Spoon	154k	2.06k	3,891	106	1.3k
Apache Maven	92.5k	1.05k	11,567	150	3.1k
Apache Spark	85.6k	1.06k	32,821	1,805	33k
Apache SkyWalking	84.7k	1.58k	7,022	397	1.9k
Jackson Core	52.3k	283	2,025	59	200
Alibaba Arthas	44.2k	586	1,726	155	29k
Jacoco	38.8k	633	1,749	46	3.2k
JUnit4	31.2k	471	2,486	151	8.3k
Google gson	25.8k	212	1,650	124	21k
SLF4J	13.5k	256	1,956	61	1.9k

1. We first curated a list of software organizations that are present on GitHub. Among the organizations we selected are Apache, Google, QuarkusIO, AWS, Alibaba, Junit-team, Mockito, ReactiveX, Spring, Facebook, Bazel build, Jenkins, etc. To which we added other organizations that deliver known software, such as JavaParser, Netty, FasterXMl, Jacoco, Qos, Inria.
2. To ensure that the software builds correctly, we focused on projects that use a build automation process that successfully passes. We selected Maven projects for this purpose. We also selected projects that support up to Java 14 included since it was imposed by the baseline tool to which we compare to.
3. We selected the most popular projects (based on their number of 'stars' on Github) and with more than one thousand commits. We finally reached 18 real-world and representative software projects: industrial and academic projects, small to very large-sized projects, monolithic and modular, and simple to complex projects.

Table 6.1 describes our curated final list of software projects.

6.2.3 Results

We empirically evaluate the key properties of the *HyperAST*. To do so, we can classify the properties in two categories. A property can either be objectively checked with a simple assertion, *e.g.,* identity of parsing and re-serializing, or needs a comparison with an existing tool, *e.g.,* for CPU/memory performance and finding references. We choose to compare the *HyperAST* against Spoon [Paw+15] as it is a well-known tool in the community, which is used to analyze

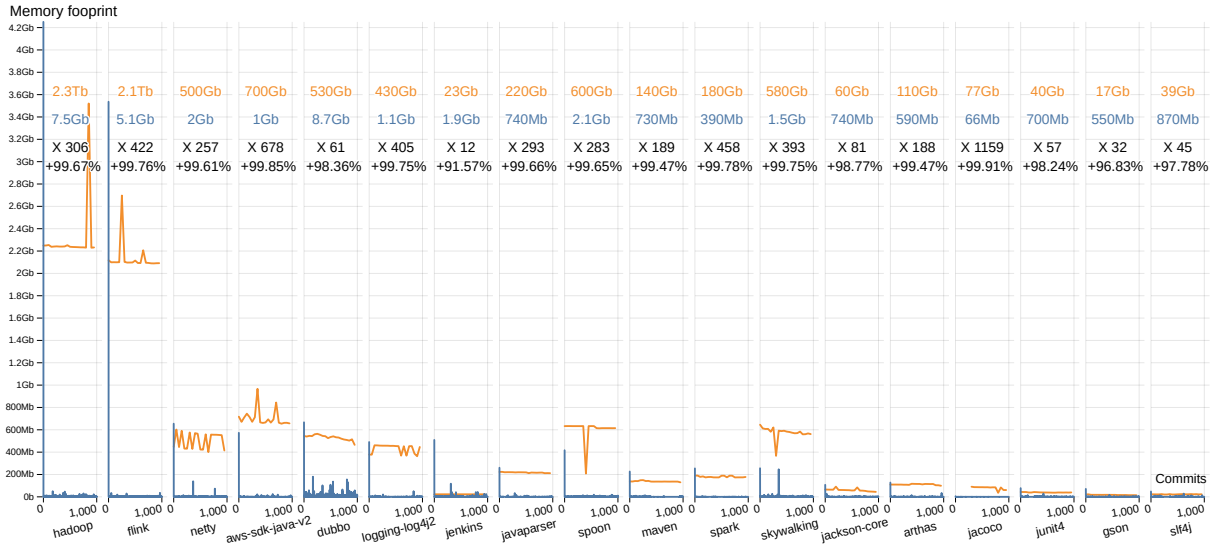


Figure 6.4 – Megabytes of memory taken for each commit, legend: orange=spoon, blue=hyperAST, with total, x order-of-magnitude difference, and + gain. Note: we do the analysis starting from the last commit published corresponding to the commit n°0.

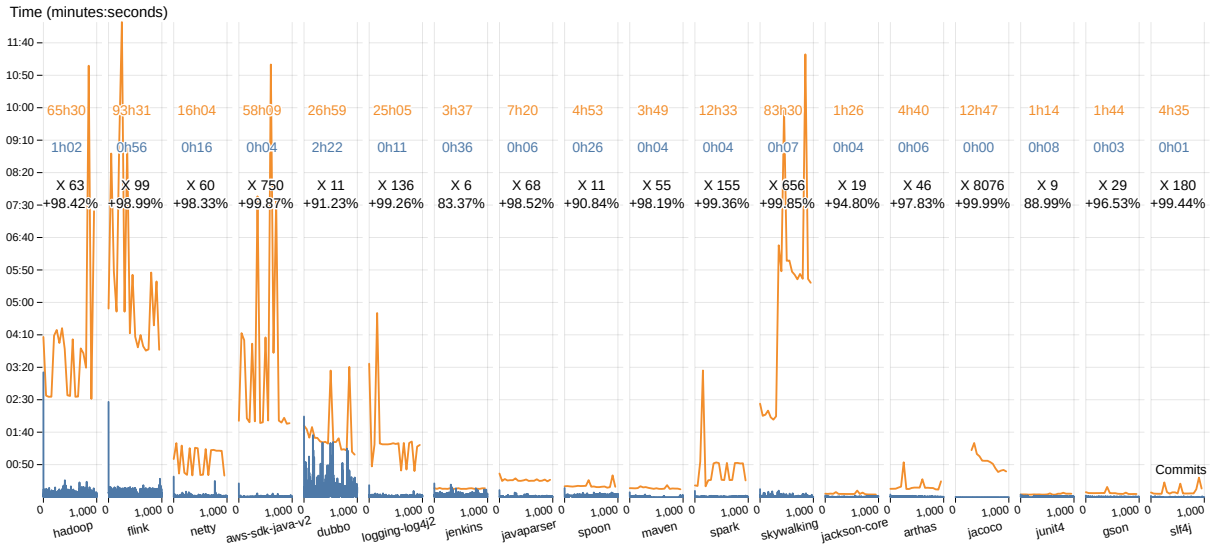


Figure 6.5 – CPU time taken to construct each commit, legend: orange=spoon, blue=hyperAST, with total, x order-of-magnitude difference, and + gain. Note: we do the analysis starting from the last commit published corresponding to the commit n°0.

and modify Java code. Thus, we observe differences of performances between Spoon and the *HyperAST* both in time and in memory for construction and finding references.

RQ1: Can we compute the HyperAST correctly over several versions?

To answer RQ1 we implemented a serialization service to parse the code and pretty-printing it back to the file level. We can thus evaluate the sound incremental construction of the *HyperAST* with the following equivalence.

$$code \equiv prettyprint(parse(code))$$

Therefore, the goal is to show that the construction of the *HyperAST* is a sound transformation of the Git Merkle DAG and the code Blobs. To do that, we performed the above verification for each project using a set of Java file Blobs uniformly sampled from the Git object database.

On a total of 299 041 Java *Blobs* from all commits, the *HyperAST* was able to parse and to serialize 299 007 them back successfully with an exact equivalence. Thus, achieving 99.98% correctness. We looked at the remaining 34 *Blobs* and found that it was caused by Tree-Sitter: these *Blobs* contain errors in the code with missing tokens, which the Tree-Sitter parser corrected by adding the most likely tokens that respect the Java grammar rules. This feature of Tree-Sitter allows the *HyperAST* to analyze even erroneous code in contrast to existing tools.

Therefore, we can answer RQ1 positively. Ensuring the sound construction of the *HyperAST* is essential for code transformation purposes, such as patch application or code refactoring.

> ***RQ*1 insights:** Results show the soundness of the *HyperAST* construction with 99.98% of correctness. It is a preserving transformation for correct code into the *HyperAST*. Erroneous code is slightly corrected at the token level in 0.02%.

RQ2: How does the HyperAST perform and scale compared to a traditional approach?

This RQ investigates the difference between the *HyperAST* and traditional approaches in terms of memory footprint and execution time for building ASTs over versions. To do so, we selected the most recent thousand of commits in each project to analyze for a fair comparison between Spoon and the *HyperAST*. For the memory footprint, we measure the heap with and without the constructed structures, *i.e.,* the *HyperAST* and the Spoon ASTs. For construction time, we measure the time in CPU cycles from the start to the end of the construction.

With the *HyperAST* we use the Rust functions $Instant :: now()$ and $Instant :: elapsed().as\_nanos()$ for time measurement and $jemalloc_{ctl}$ library for memory measurement. With Spoon, we use the Java function $System.nanoTime()$ for time measurement and the functions $runtime.totalMemory()$ and $runtime.freeMemory()$ for memory measurement.

Figures 6.4 and 6.5 give the measured memory footprint and the construction time for both Spoon and the *HyperAST*. Note that since we started by treating the latest commit in the history and went back in time to treat the other commits, the depicted figures show the results in that order of commits, *i.e.,* left (0) being the most recent commit. Overall, the *HyperAST*

outperforms Spoon. We observe less memory footprint and less construction time for *HyperAST* than Spoon.

Regarding the memory footprint, the *HyperAST* outperforms Spoon in all projects, as shown in Figure 6.4. On minimum and maximum, respectively, the *HyperAST* consumed 63 MB in Jacoco and 7.5 GB in Hadoop of memory compared to 17 GB in Gson and 2.3 TB in Hadoop for Spoon. We observed the smallest and the biggest gains in memory footprint of +91.8% and +99.9%, respectively, in the Jenkins and Jacoco projects. The order-of-magnitude difference in memory footprint between the *HyperAST* and Spoon varied from ×12 in Jenkins up to ×1159 in Jacoco. Note that the sudden drop in the Spoon project is due to erroneous code in the commits that could not compile, whereas the *HyperAST* can still handle them thanks to the robustness and resilience of Tree-Sitter.

Regarding the construction time, we also observe benefits for the *HyperAST* over Spoon, as shown in Figure 6.5. The *HyperAST* outperforms Spoon for the analyzed projects. On minimum and maximum, respectively, the *HyperAST* took 1 min in Slf4j and 2 h and 22 min in Dubbo of construction time, compared to 1 h 14 min in Jacoco and 93 h 31 min in Flink for Spoon. We observed the smallest and the biggest gains in construction time of +83.4% and +99.99%, respectively, in the Jenkins and Jacoco. The order-of-magnitude difference in construction time between the *HyperAST* and Spoon varied from ×6 in Jenkins up to ×8076 in Jacoco. It is worth noting that the *HyperAST* outperforms Spoon even thought we run it on a single CPU core (no parallelization) in contrast to Spoon with the JDT. Thus, we actually consumed less computing power per commit. Parallelization would improve the construction time, *i.e.,* latency, but, is left as future work.

Figures 6.4 and 6.5 confirm that the memory footprint and the construction time of the *HyperAST* are always paid initially and then are much lower for later commits. Therefore, the *HyperAST* depends on the code size only for the first commit and then only depends on the commit size, *i.e.,* code changes. For Spoon, it is rather dependent on the code size as expected, where the larger the code is, the more memory and time it consumes to construct its AST.

Finally, we clearly see the issue of scalability with Spoon only on a thousand of commits. However, a thousand of commits did not stress test the *HyperAST*. To do so, we took the Hadoop project that showed to be the most costly to build, and we built the *HyperAST* on all its history of 25 749 commits. The measured memory footprint was 71 GB and the construction time was 18 h 21 min. Compared to Spoon with a thousand of commits, we are still outperforming it with gains of +72% in construction time and +97.4% in memory footprint. Resulting in an order-of-magnitude difference of ×3.6 for construction time and ×38 for memory footprint.

***RQ2* insights:** The *HyperAST* shows significant results in performance compared to Spoon. It provides scalability gains in memory footprint from +91.8% to +99.9% and in construction time from +83.4% and +99.99%. Same observation holds when the *HyperAST* takes all the 25 749 commits of Hadoop.

6.2.4 Threats to Validity

Internal Validity

Considering performance measurements, we choose to measure CPU and memory consumption for each processed commit. Our goal was to show to what extent the *HyperAST* scales on complex real-world software with large histories while quantifying it per commit to observe the scalability tendencies. Measuring once for all commits would not have accurately reflected variations of analysis costs, as during development a software might change in quality, complexity and size. Measuring in finer grain than a commit, say for a module, a package or a class, may also be biased, as the measurements could overweight the actual processing cost itself. Thus, measuring performance per commit gives more confidence in the results.

Moreover, as spoon compiles the code, some commits could not be analyzed with Spoon. Whereas, we still were able to analyze them and insert them in the *HyperAST* thanks to the resilience of Tree-Sitter for ill-formed code. In addition, for the validity check with precision and recall, we had to select the commits that Spoon successfully compiled on which it computed the association tables. Thus, being able to compare the found references on both approaches. Our goal here was to show that the *HyperAST* can be used in a code analysis efficiently. However, we did not measure the effort of developing the service of finding references, as this was not the goal of the present contribution. Nonetheless, this is left for future work.

External Validity

We implemented and evaluated the *HyperAST* approach for Java with maven build system. Our conclusions in theory could generalize to other programming languages with similar features than Java (*e.g.,* strong static nominal typing). Nonetheless, further experimentation remains necessary on other languages to generalize our results. Note that Tree-sitter is able to support other languages (*e.g.,* C, JavaScript). Thus, making the *HyperAST* extensible beyond Java by integrating the Tree-sitter grammars of other languages. However, the goal of this contribution was not to support multiple languages but to show the scalability benefits provided by the *HyperAST*. Since the publication of [Le +22] we added support for C/C++ without much issues (some expression where missing from the tree-sitter grammar of C/C++, *e.g.,* inline asm) and it successfully handled the Linux kernel and Stockfish, yet we still lack a more systematic benchmark for example comparing the *HyperAST* to Cocinelle [LM18] pattern matching or Tree-Sitter queries (both would require us to implement their query language for the *HyperAST*). Moreover, we relied on Spoon as a baseline, mainly due to its popularity as a frontend to the Java development tools (JDT). Hence, we cannot generalize the observed benefit of the *HyperAST* compared to Spoon for other AST tooling, such as JavaParser as it also resolves the references. This is left for future work to enhance the comparison results.

Finally, the *HyperAST* considers Git histories. As discussed in this chapter, the *HyperAST* can operate on any snapshot-based history similar to Git, with minimal development (see implementation chapter 9.1). For the other cases (e.g. SVN), a classical technique consists in converting a repository to Git similarly as Software Heritage [DZ17] initiative does before

archiving repositories.

Conclusion Validity

Our evaluation gave promising results, showing that the *HyperAST* allows to save a lot, respectively, with an order-of-magnitude difference from 6 up to 872 in construction time and from 12 up to 1158 in memory footprint. The evaluation results also showed providing a reliable service for finding references with an average precision and recall respectively of 90% and 97%. Even though we evaluated it on 18 projects, we plan to further extend the scope of this evaluation. Indeed, including more projects would give more insights and statistical evidence. Yet, our curated list of projects represents real-world complex software with very large histories.

6.3 Conclusion

This chapter presented a novel type of AST, namely the *HyperAST* that allows temporal code analyzes to scale across large code histories. The evaluation on 18 large projects shows that the *HyperAST* is able to scale to thousands of commits with multiple order-of-magnitude difference between the *HyperAST* and Spoon, from $\times 6$ up to $\times 8076$ in construction time and from $\times 12$ up to $\times 1159$ in memory footprint. While the *HyperAST* requires up to 2 h 22 min and 7.2 GB for the biggest project, Spoon requires up to 93 h 31 min and would require 2.2 TB to maintain all these ASTs in memory. The gains in construction time varied from 83.4% to 99.99% and the gains in memory footprint varied from 91.8% to 99.9%. These benefits and gains were also observed when we built the *HyperAST* on the whole 25 749 commits history of the Hadoop project in contrast to only a thousand of commits with Spoon.

Chapter 7

(Spacial analysis) Efficient Reference Analysis on Code Histories

This chapter discusses how to use the *HyperAST* and its metadata for a classical task: navigating in the code from a given declaration to all its references. This is a central functionality used to implement multiple well-known approaches, such as code smells detection [Tuf+17], bug prediction [Pal+17], refactoring [TKD20], and co-evolution [Le +21; LY17]. Furthermore, evaluating this use case will indirectly show the performance and correctness benefits of the *HyperAST* (section 7.2). This analysis focuses on Java source code as it is widely used and similar to many other programming languages. Furthermore, it presents challenging name resolution, as presented in section 3.1.2, Poulsen, Zwaan, and Hübner call them multi-phased type (or name) resolutions [PZH23]. Finally, given the complexity of the referential semantic of Java, it should be possible to adapt our implementation to other programming languages.

> The approach and results presented this chapter were partly published at ASE 2022 [Le +22]. This chapter comes with more details on the inner working of the reference analysis, and a supplementary research question (RQ4).

7.1 Approach

Given a declaration, finding all its references is a standard feature in most code analysis tools. It often implies maintaining an association table between declarations and references. It works fine in the case of a single version. However, maintaining such global tables and ASTs for multiple versions requires heavy maintenance work to keep those numerous tables synchronized. Instead, our solution rather uses an oracle capable of answering with certainty when a reference is absent from a given piece of code, *i.e.,* from a subtree. With such oracle only the subtrees that may contain references need to be explored, while the rest is pruned from the search. The interface to our oracle is modeled as a function that takes a signature and returns *Absent* if there is no reference corresponding to the given signature in the associated subtree, otherwise it returns *MaybePresent*. Relaxing one side of the oracle semantic has multiple advantages that will be detailed during this chapter.

The preprocessing phase takes place during the building of the *HyperAST*. As described in

chapter 6, metadata can be computed and persisted in each node of the *HyperAST* between the deduplication and insertion step, such that compiling metadata is made incremental. The metadata computation consists in partially resolving references while propagating references and declarations (they are also cached), the remaining unresolved references are the one actually compressed and stored as metadata.

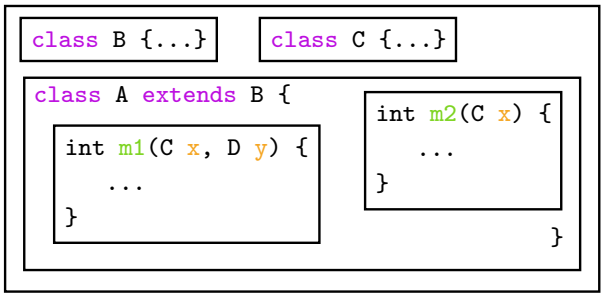


Figure 7.1 – Example source code represented as nested subtrees.

7.1.1 Representing a signature

Naturally to search and match signatures we need to choose a representation. Listing 2 show how signatures for references and declarations are represented. Here, we focus mainly on simple qualified identifier, *i.e.,* a chain of identifiers. Yet depending on the chosen semantic of references, additional types of signatures are defined, such as calls, array accesses, primitives, `this`, `super`. For example, in Java `import java.lang.Object;` correspond to `Type(Identifier(Identifier(Root,"java"),"lang"),"Object")`.

```
enum Sig<Id> {
    // resolved signature
    Root,
    // unresolved signature
    Pending,
    // scoped identifier
    Identifier(Id,String),
    // scoped type
    // needed because of obscuring
    Type(Id,String),
    // Choice between sig. variants
    // needed because of shadowing
    // produced by partial resolutions
    Choice(Vec<Id>),
    // language specific cases: this, primitives ...
}
```

Listing 2 – Construct used to define a signature

For performance reasons (mainly memory consumption) sets of references and declarations are actually backed by a store that deduplicates signatures and their components, *i.e.,* the signature

store is a DAG. For example, given `java.lang.Object` and `java.lang.Float`, `java.lang` will be shared. This nesting is actually easy to apply in combination to the store by properly setting the generic type for `Sig` in listing 2 as reference to a signature in the store. Listing 2 shows how it can be achieved easily.

7.1.2 Indexing References

Our indexing leverages the extensible metadata provided by an *HyperAST*. Concretely, the metadata serve the purpose of an index, given a specific search query, it accelerates the access to searched code elements. More specifically the goal of the indexing is to provide an oracle in each node of the *HyperAST*, and prune subtrees that will surely not provide results for the given search. Consequently, by default it is always functionally correct for an oracle to return *MaybePresent*.

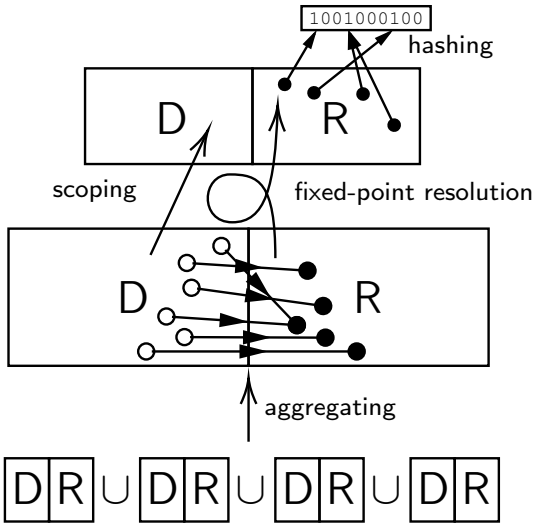


Figure 7.2 – Partially resolve references and compute oracles from still unresolved references.

Figure 7.2 illustrates both the principle of the partial reference resolution and the computing of a reference oracle. Computing a reference oracle on a subtree is done in post-order traversal *i.e.,* a node is processed once all children have been processed. It is an incremental algorithm that takes advantage of the deduplication of subtrees to only process each identical subtree once. Here each subtree has both a set of declarations **D** and a set of references **R**.

The Extraction and Aggregation Process

This process vary depending on the type of the node being processed and the one being aggregated. For example, given the assignment statement $Cy = x$; the expression x is handled first, x is extracted as $Identifier(Pending, "x")$ and inserted in the set of references. Then back to the statement, the type access C is extracted as $Type(Pending, "C")$; the local variable declaration y is extracted as $Identifier(Root, "y")$ (of type C). C and x (aggregated) are inserted in the set of references, $y \rightarrow C$ is inserted in the set of declarations.

Listing 3 illustrate the accumulation of each child state to the current partial reference analysis state. Where a state S contains a set of visible declarations and a set of unresolved references but also other language specific information, for example to handle visibility of class members.

```
fn accumulate(current: S, child: S) -> S {
    // ...
    else if current.is_type_body() {
        match (current, child) {
            // ...
            (Members(mut l), FieldDeclaration{
                visibility,
                kind,
                name,
            }) => {
                let identifier = Sig::Identifier(Maybe, name);
                // ... // additional instructions to leverage signature store
                l.push((visibility, i, kind));
                Members(l)
            }
            (Members(mut l), MethodDeclaration{
                visibility,
                kind,
                name,
                parameters,
            }) => {
                let identifier = Sig::Invocation(Maybe, name, paramters);
                // ... // additional instructions to leverage signature store
                l.push((visibility, i, kind));
                Members(l)
            }
            //...
        }
    }
    // ...
}
```

Listing 3 – Combine partial referential analysis states

For example, taking fig. 7.1, when processing the body of `class A`,

— initializing the state corresponding to the body, such that $s_{body_0} = Members(\emptyset)$,

— then combining the state obtained from `method m1` we name s_{m1}, we have $s_{body_1} = accumulate(s_{body_0}, s_{m1})$, such that now $s_{body_1} = Members(\{m1(C, D)\})$,

— then combining `method m2` we name s_{m2}, we have $s_{body_1} = accumulate(s_{body_0}, s_{m2})$, let us consider that $m2$ is `public`, such that now $s_{body_2} = Members(\{m1(C, D), public\ m2(C)\})$ (visibility is important for name resolution).

The Partial Resolution Process

This process operates on the set of declarations and the set of references. The partial resolution is a fixed-point algorithm *i.e.,* **fixed-point resolution**. To be more efficient, declarations are resolved first. For example, given a set of declarations only containing the declaration $Identifier(Root, "y")$ of type $Type(Pending, "C")$, $Type(Pending, "C")$ cannot be resolved for now (thus y keep the same type), while $Identifier(Pending, "x")$ would be resolved to $Type(Pending, "C")$.

In post order, the *aggregating* of (D,R) consists in reinserting declarations and references in a single (D,R) tuple (if relevant, already initialized with the current declaration or reference). Then, *scoping* declaration allows qualifying declaration relative to their parents. After that, the *fixed-point resolution* allows resolving references using declarations. Resolutions are cached, which allows implementing the fixed-point condition. Fully resolved references can be removed. Finally, *hashing* creates the reference oracle using a bloom filter, as explained in the next paragraph.

> A major subtlety in most languages, is that **Identifier**s should in priority match variable declarations and then type declarations (*i.e.,* Obscuring).

Structure of Oracle Persisted in *HyperAST*

To instantiate our oracle we rely on Bloom filters [Blo70]. Bloom filters are notably used in network security [GA13; BM04], and bio-informatics [MP11]. Multiple alternatives and variants of Bloom filters exist in the literature and libraries depending, we use the original Bloom filter for its simplicity and effectiveness [Blo70], as it also facilitated further specializations.

A Bloom filter is a probabilistic data structure efficient to check whether an element is absent from a set. Consequently, checking for the presence of an element only has probabilistic guarantees. This probabilistic relaxation allows to gain a substantial space advantage over other data structures, as elements are not required to be stored. Indeed, in the usual hash sets, to counteract hash collisions, inserted elements must be kept to be compared with new elements. Concretely, a bloom filter is an array of n bits all initially set to 0 and a set of hash functions H. Then, for each element, the Bloom filter use those hash functions to get a set of integers indexes. They set to 1 the corresponding indexes (modulo n) in the bit array to indicate the presence of element. The number of hash functions is usually a major optimization point where the false positive rate (FPR) is function of n, the number of hash functions, and number of expected inserted elements. Growing a Bloom filter can be a major issue (also removing elements), as it requires to re-insert all elements, thus, in part, defeating the compression benefits.

In practice, as metadata for each subtree in an *HyperAST*, our oracle stores the unresolved references obtained through our partial reference resolution. Therefore, to instantiate each Bloom filter we have four interacting values to consider: the FPR, the number of bits n, the number of hash functions $|H|$ and the expected number of unresolved references r. Notice, that our oracle is used in a recursive data structure, thus, during a traversal the FPR is compounded, meaning that we can reduce n and $|H|$ significantly without impacting the overall effectiveness on the approach. Moreover, we are building each oracle from a set of references known at instantiation

(due to the non-contextual post order traversal), therefore, we know the number of elements that will ever be stored inside each Bloom filter. Finally, to reduce the computing cost for the happy path we would like to keep the number of hash functions as low as possible, actually, using a single hash function per size of bitset (due to modulo) also keeps the code simpler as we can hard code it (or choose at compile time). So, with a single hash function we should have FPR $\approx \frac{|r|}{n}$, *e.g.,* with $n = 2|r|$ each oracle would prune subtrees that do not hold our reference with a rate of $\frac{1}{2}$.

A few optimizations are also possible:

— n can be chosen in powers of 2 to make the modulo a shift operation.

— A lower bound can be chosen given the number of unresolved references, the size, or height of the subtree.

— An upper bound can be chosen on similar parameters. In both cases, at both extreme the efficiency of our oracle diminish. As it goes down in a subtree each pruning only removes a few elements, while it goes up, it has fewer chances of not containing a particular reference.

— The upper bound could also lead to a kind of multi-staging, where we choose hash functions with more collisions, *i.e.,* not taking full signatures but just names, *e.g.,* `java.lang.Object` produces a list composed of "java", "lang" and "Object". This last optimization has not been implemented yet.

7.1.3 Finding all References Matching a Signature

This searching phase consists in going done the code (subtrees) to find references corresponding to a given signature. Without using the oracles, it would be very similar to a naive traversal trying to match a given pattern.

Figure 7.3 illustrates the actual search process. Given the reference E, it recursively goes down the AST. At each subtree, the local reference oracle is used to check whether the subtree might contain E, if it tells that E is absent then the whole subtree is skipped, thus saving operations that would not lead to finding any wanted references.

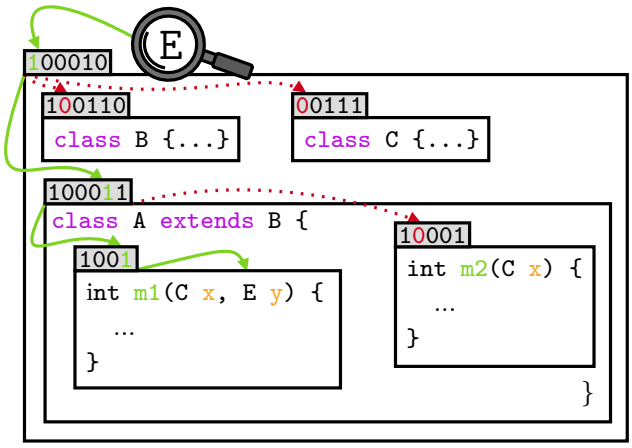


Figure 7.3 – Example ref ana using oracles.

Data: a node $n \in T$, and a signature s
Result: R the set of nodes from T matching s,
in itself the function returns a set of aliases of s

```
1  if ref_oracle(n, s) = Absent then
2      return ∅;
3  if let "package" p ";" = n then
4      if object(s) = p then
5          return {name(s)};
6  else if let "import" o ".*;" = n then
7      if object(s) = o then
8          R = R ∩ n;
9          return {name(s)};
10 else if let "import" c ";" = n then
11     if s = c then
12         R = R ∩ n;
13         return {name(s)};
14 ...
15 else
16     if n is variable access ∧ s = n then
17         R = R ∩ n;
18     else if n is type ref ∧ s is type ref ∧ s = n then
19         R = R ∩ n;
20     ...
21     A = ∅;
22     foreach c ∈ children(n) do
23         A = A ∩ search(s, n);
24         foreach a ∈ A do
25             A = A ∩ search(a, n);
```

Algorithm 1: $search(s, t)$

Algorithm 1 presents a pseudocode of the search algorithm. The first condition allows to stop the search in the current subtree when the reference oracle (ref_oracle) guarantee there is no reference in n matching s. Then, it presents three special cases specific to Java, that alias the signature s (line 3 to 13): package declaration, star import, import. Otherwise, (line 16-20) we try to match the signature s to the one represented by the current node n, *e.g.,* when it is a variable access or a type reference. Finally, (line 21-23) for each child it recursively searches for s and other aliases a while adding newly found alias to A.

7.1.4 Finding all References from a Declaration

Given a declaration *i.e.,* a piece of code, to find all references to this declaration we need to climb up in the tree of code up until the declaration is not visible anymore. Once reaching each new scope we search for references with the approach presented in the previous section (section 7.1.3). It starts from the given declaration and goes up in the AST while the declaration is visible, each time the scope changes a search is done in the subtree.

7.2 Evaluation

This section presents the evaluation of the *HyperAST* used to efficiently find all reference to given declarations. First, we present the research questions to then discuss the results. Then, we present the data set and evaluation process. We finally discuss the threats to validity, limitations, and the scope of the approach. **All the material of this section and a replication package are available on our companion web page** [1].

We ran the implementation of our approach on the following hardware configuration: *2 x Intel(R) Xeon(R) Gold 6238 CPU @ 2.10GHz; 187Gb ram; 1 T SSD*, running *Ubuntu 18.04.6*.

7.2.1 Research Questions

We now formulate the research questions as follow:

RQ1 **How does the *HyperAST* perform and scale on the code analysis task of finding references compared to a traditional approach?** This aims to investigate whether a code analysis task can leverage on the *HyperAST*, thus opening a new perspective to work with the evolution/time dimension rather than on only a single version at a time.

RQ2 **To what extent our approach decorrelates the cost of searching for references from the size of the code base?** This aims to investigate the correlation between the number and types of declarations against the cost of finding their corresponding references. This research question also demonstrates that we need neither association tables nor searching inefficiently in the whole code base. It actually details results shown in RQ2 by specifically looking at the cost of each individual search, while also detailing the context where the search is done *i.e.,* code size.

[1] https://github.com/quentinLeDilavrec/ASE2022

7.2.2 Data Set

We use the same dataset as section 6.2.2.

7.2.3 Results

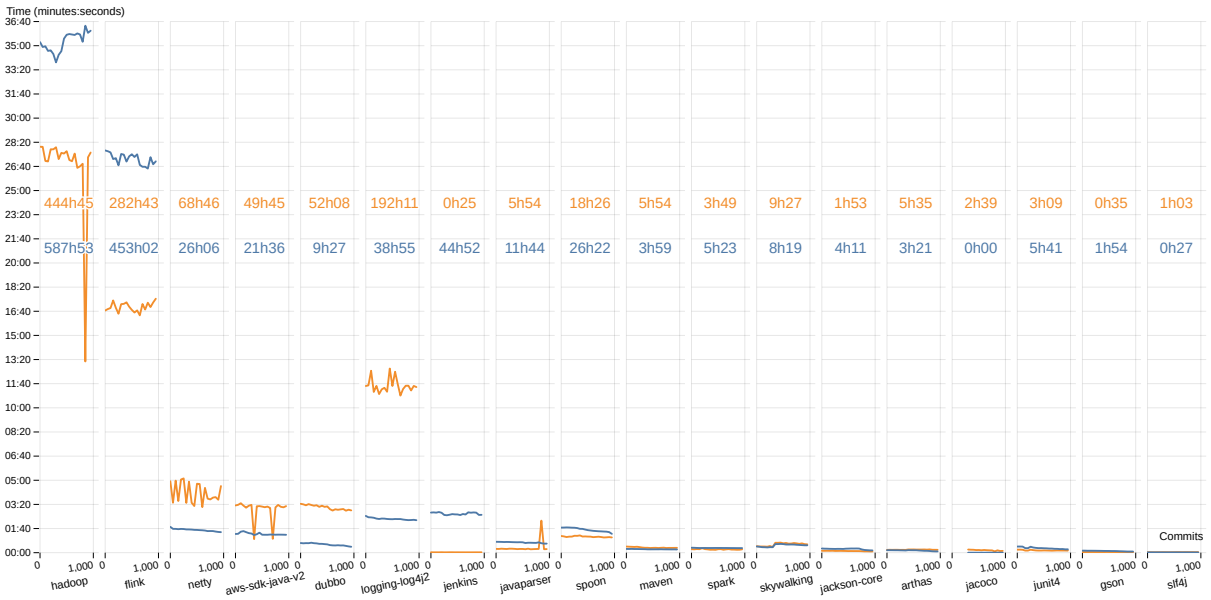


Figure 7.4 – CPU time of finding references for each commit, legend: orange=spoon, blue=hyperAST, with total.

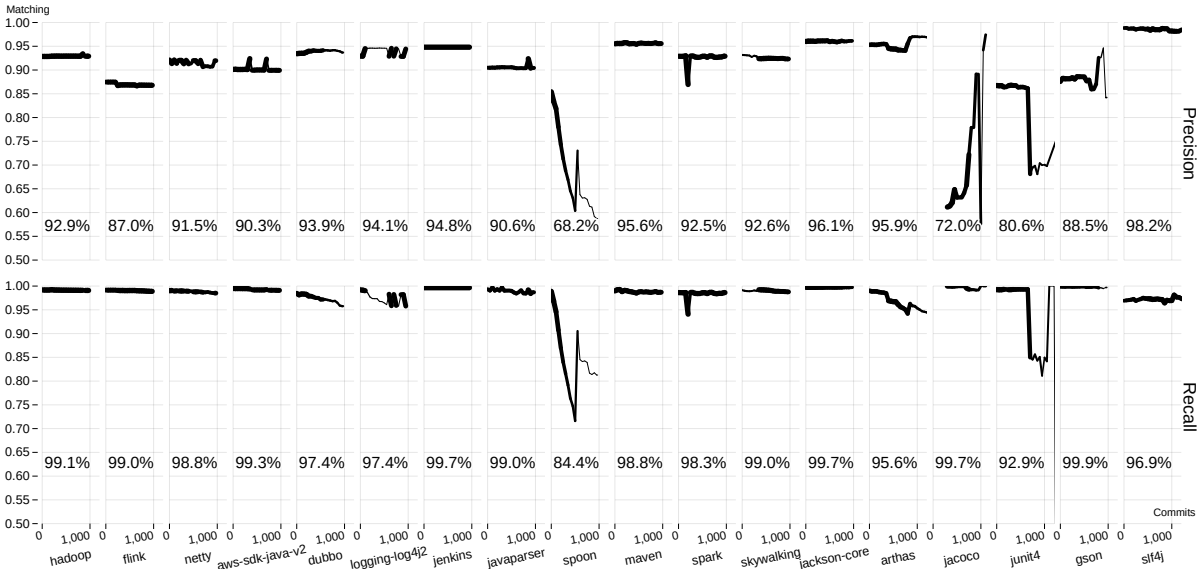


Figure 7.5 – Validity of finding references with precision (top) and recall (bottom) per commit, with respective average.

RQ1: comparing our approach against the usual association table approach

This RQ aims to compare the *HyperAST* and Spoon on the same task of code analysis, namely finding references of declarations. For the *HyperAST*, we use our prototype solution based on our reference oracle. For Spoon, we rely on its constructed association tables.

We first check the validity of our prototype solution. Then, we compare the performances of finding references. We consider Spoon as a ground truth. Thus, we can compute *precision* and *recall* for the *HyperAST* in finding references. Precision and recall vary from 0 to 1, *i.e.,* 0% to 100%. They are defined as follows:

$$precision = \frac{ResolvedReferences \cap ExpectedReferences}{ResolvedResolutions}$$

$$recall = \frac{ResolvedReferences \cap ExpectedReferences}{ExpectedReferences}$$

The comparison of references is not based on their labels, as this is not a guarantee of finding the same references. Rather, the comparison is based on the *start* en *end* characters in the code. This is a more restrictive comparison that further would discriminate our approach while increasing confidence in the results.

Figure 7.5 shows the different measured precision and recall in our case studies for the *HyperAST*. The measurements are per commit where the left (0) represent the most recent commit as we went back in time up to the thousand commit. We observed on average 90% precision and 97% recall. Precision ranged from 68.2% to 98.2% in Spoon and Slf4j projects, while recall ranged from 84.4% to 99.7% in Spoon and Jenkins projects. In the projects where precision was low, namely Spoon, Jacoco, and Junit4, we investigated the found references by the *HyperAST* and Spoon. We found that many discrepancies are due to minor shift in the start and end characters, mainly due to the inclusion of comments to the found references. However, other cases were due to an over-estimation of the *HyperAST* for possible shadowed or overridden references.

Figure 7.4 shows how the *HyperAST* compares to Spoon in terms of searching time to find all references. It shows that it is similar on average to Spoon for small-sized projects up to 180 k of **LOC!** (**LOC!**). We observe that our prototype of finding reference over preforms for medium-sized projects up to 300 k of **LOC!** and it under performs for large-sized projects with millions of LOC. However, in the Haddop and Flink projects, Spoon out performs the *HyperAST* by roughly 10 min per commit. Nonetheless, in contrast to Spoon that is only capable of calculating all references for all declarations at once, the *HyperAST* can calculate the references for a single declaration at a time whenever needed. Therefore, depending on the usage scenario, the *HyperAST* will outperform Spoon in case of finding references for some given declarations and not all of them.

***RQ*1 insights:** Results show the validity of finding references based on the *HyperAST*. We observed an average 90% precision and 97% recall without a significant difference in search time but a difference of 10 min in large-sized projects.

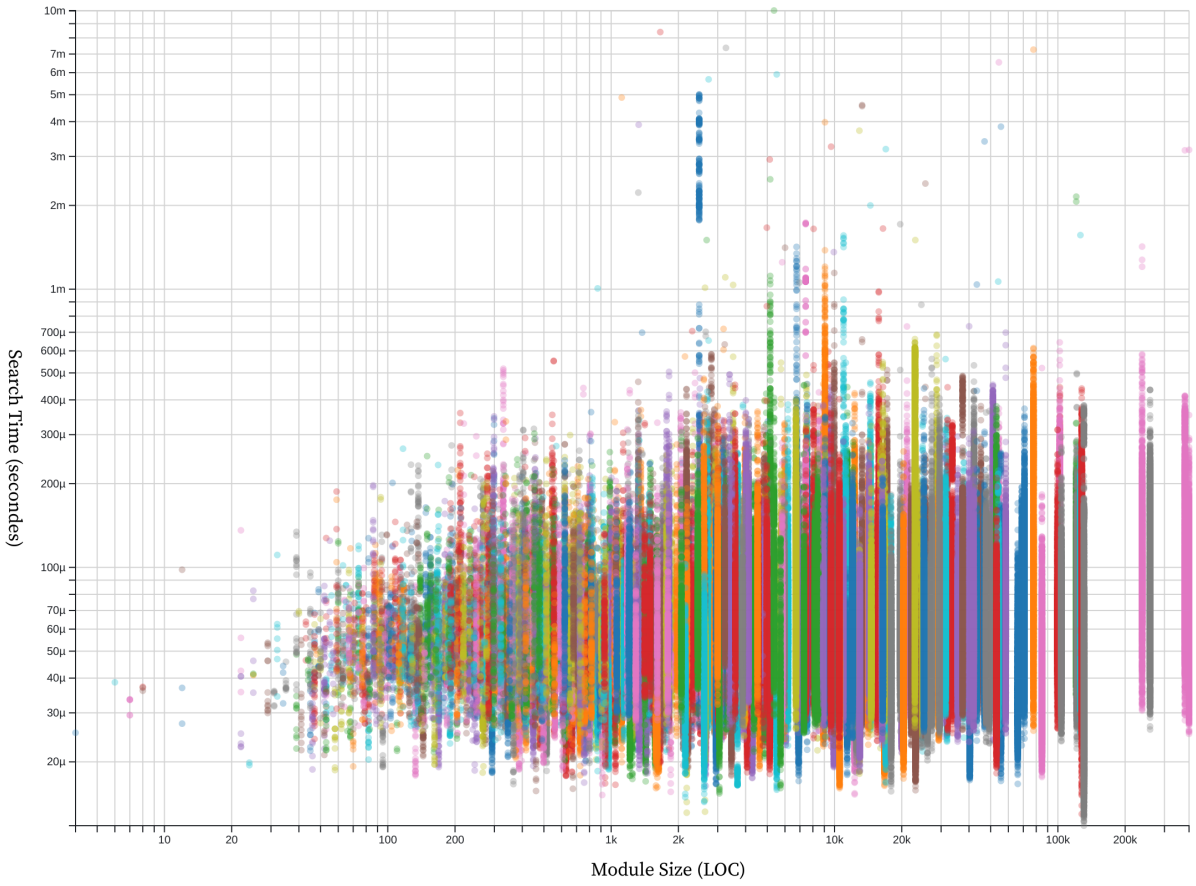


Figure 7.6 – Time cost searching for all references to local declarations (local_variable, parameter_declaration).
Each search is normalized, the search time is divided by the number of found references. The dots colors are there to help with distinguishing modules.

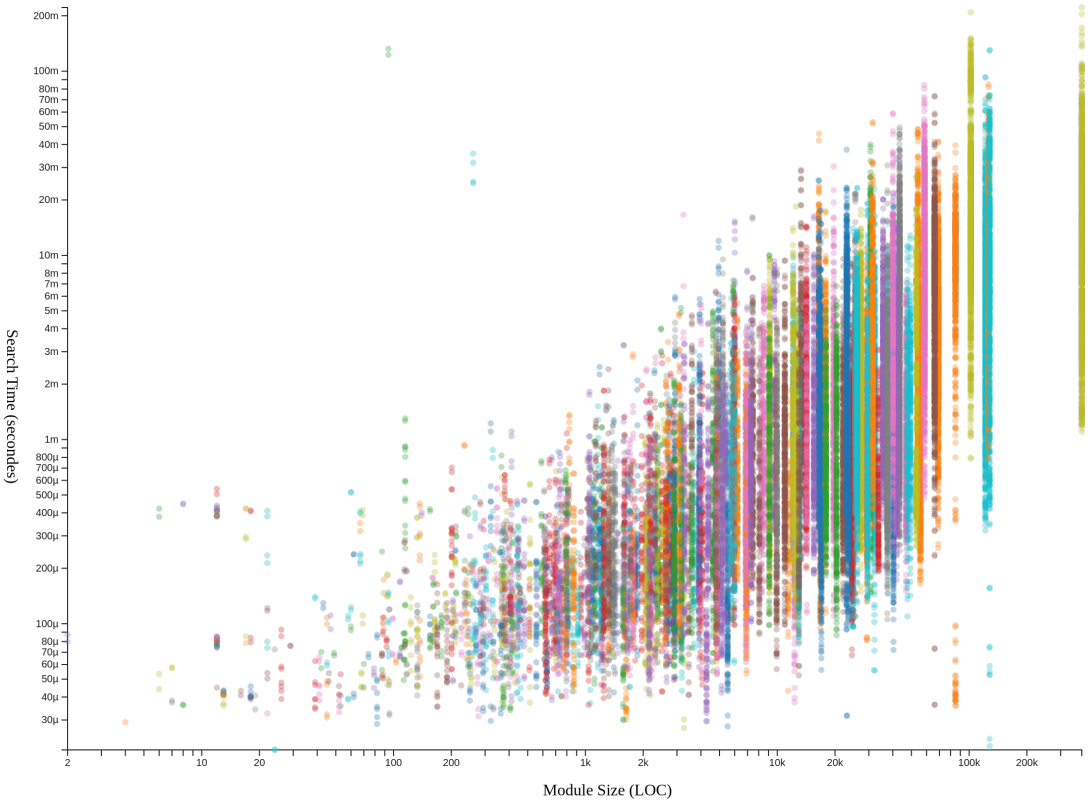


Figure 7.7 – Time cost searching for all explicit references to global declarations (class_declaration, interface_declaration).
Each search is normalized, the search time is divided by the number of found references. The dots colors are there to help with distinguishing modules.

RQ2 Lazy def-use resolution

Contrary to traditional def-use computations, our lazy search do not need to go over the entire code just to find the references of a few declarations. With our approach the cost of computing def-use should be proportional to the number of found references, while disconnected from the size of the codebase. One major usecase would be to just search for references to changed delcarations. Considering the lazyness of our approach *w.r.t.* declarations, in Figure 7.4 we show the expected cost of finding all references to 50% and 10% of the declarations. In this use-case, we were always faster and less costly than Spoon. For the largest projects, we decreased the search time to roughly 17 and 3 minutes in Hadoop and to roughly 13 and 2 minutes in Flink, compared to their respective 26 and 17 minutes.

Figure 7.7 presents performances due to the major specificity of our approach. It shows the average time it costs to find a reference of a given type declaration over the size of the considered code (measured in Java LoCs, computed with `scc`,[2] ignoring blank lines and comments). We can observe a correlation between the search duration and the size of the code. Thus, our approach does not completely decorrelate the time cost from the size of the code. However, in the worst case the computation time grows linearly with the size of the code (1ms at 1KLoC, 10 ms at 10KLoC, 100ms at 100KLoC).

On contrary as it should, in Figure 7.6 for local def-use, we do not observe any significant correlation between search time and the number of LoC in the analyzed module.

Finally, using these results to estimate the cost of resolving a set of declarations having a total number of a 1000 references in 20KLOC, it should take less than 20 seconds.

***RQ2* insights:** This detailed presentation of performances results shows the lazy nature of our approach. Indeed, our approach obtains performance benefits searching a **subset** of declarations, here, a single one each. Thus, comparing to Spoon on Figure 7.4, searching for 50% and 10% of the declarations and references will divide our overall search time by half and ten. Thus, making it faster than spoon on realistic searches that focuses on specific declarations.
For global declarations, the worst case is linearly correlated to the size of maven modules, whereas moving to the best case de-correlates the search cost from the module size. It shows the difference between successful and unsuccessful uses of oracles.

7.2.4 Limitations

It would require further evaluations to precisely quantify the impact of each hyperparameters on the performances of our approach.

[2] https://github.com/boyter/scc

7.3 Conclusion

This chapter presented an application of the *HyperAST* to the resolution of referential relation, more specifically, given a declaration finding all its references. This is a major component of a static impact analysis as presented in chapter 5. This analysis focused on Java source code as it is widely used, similar to many other programming languages and present multiple name resolution challenges.

In the evaluation, we compared our approach to the standard association table using Spoon to populate it. We observed on average 90% precision and 97% recall when comparing exact reference relations. For the search time, when resolving all reference relations we did not find a significant difference. Yet compared to the usual approach that requires to rebuild the association table by going through each reference, we were able to show that by the lazy nature of our approach that each declaration could be searched individually, in the worst case in 200 milliseconds.

Chapter 8

(Temporal analysis) HyperDiff: Fine grained tree diffs of entire code histories

This chapter presents another use case for the *HyperAST*, this time focusing on the temporal aspect of source code analysis. Indeed, as software evolves quickly, one way to study its evolution is through computing source code *diffs* on its AST representation (also called an edit script). It is composed of actions (*i.e.,* changes) that represent the applied changes from an original version to an evolved version of the code. Actions are either atomic or composed. An atomic action can be either an *insert (add), delete (remove), or update* of AST nodes. A composed action is an action composed of other actions. For example a *move* action is composed of a delete and an insert (possibly also an update), which moves a given node (deletion) to another place in the AST (insertion).

Moreover, jointly analyzing and computing the structured difference on thousands commits corresponding to hundreds thousands LoC faces scalability issues. Mainly because of the cost of: 1) parsing the original and evolved code to produce two ASTs, and hence, doing so on thousands of commits; 2) wasting resources by not reusing intermediate computation results that could be shared among versions; 3) unsuitable memory layout of compared trees by allocating nodes in the global heap (*i.e.,* not contiguous, indexed by generic pointers, etc.). Many existing works proposed to compute structured diffs (see section 2.1.3), proposing various improvement over their predecessors.

Two existing research tools are widely used to compute AST diffs, namely *Change Distiller* and *GumTree* (see section 2.1.3). Both approaches are extensively evaluated in the literature, and they are both based on Chawathe's algorithm [Cha+96], where there is a mapping phase and an edit-script construction phase. We specifically choose *GumTree* [Fal+14] as it is the latest diffing tool and has a very accessible code base, with benchmarks and implementation of previous stat-of-the-art techniques. The *GumTree* approach notably focuses on scalability by reducing its algorithmic complexity to $O(n^2)$ compared to state-of-the-art techniques, *i.e.,* $O(n^3)$. This chapter will also show that the *GumTree* algorithm is particularly compatible with the *HyperAST*. However, *GumTree* was only evaluated on pairs of files, and still suffers from scaling issues on large software projects where code tree are made of millions of nodes. Actually, there exists an implementation combining *GumTree* with the *Spoon* parser to compute diffs on commits, yet it suffers from the above-mentioned limitations *w.r.t.* scalability. To the best of

our knowledge, we are the first to propose an approach that addresses the scalability issue of computing diffs on large code histories.

The *GumTree* approach introduced a two phased mapping algorithm that trades some optimality for a lot of efficiency. An illustration of the gumtree algorithm is presented on the left side of fig. 8.1. The first mapping phase is the **top-down matching** (also called subtree matching in [Fal+14]). It is a top-down traversal (breadth-first) that consists in matching subtrees in rounds (level by level starting from the root). Each round, unmatched subtrees are opened, *i.e.,* the root of the subtree is skipped, and its children are made available to be matched on the next round. The second mapping phase is the **bottom-up matching**, a bottom-up traversal (post-order) that matches previously unmatched tree nodes using an optimal matching algorithm and the previously matched subtrees. In post-order, unmatched nodes are compared to candidates (*i.e.,* other nodes that share mapped descendants) in the other version. Then, the most similar candidate is selected to apply an optimal matching algorithm, such as RTED [PA11] or *ZS* [ZS89] (*ZS* is the default).

Note for later that *GumTree* uses and computes the following metadata on each node:

— the *height* of a subtree (in number of nodes);

— the *size* of a subtree (in number of nodes or in the number of characters);

— the *structural hash*: the hash of a subtree that depends on the type and order of nodes (ignore labels);

— the *label hash*: the hash of a subtree that depends on the type, label, and order of nodes

Note that while *GumTree* was originally evaluated on diffing files, we target diffing at the commit level, which poses the challenge of making it scale to tree multiple orders of magnitude larger (millions instead of thousands).

In this chapter, we propose a novel code differencing approach that enables the production of diffs/edit scripts at scale on large software codebases. We combine concepts of *GumTree* [Fal+14], a mainstream code differencing algorithm, and *HyperAST* [Le +22], a novel representation of code histories, to propose an incremental and memory efficient approach. In particular, rather that having an AST for each version to analyze, the *HyperAST* leverages code redundancy in space and time using a Direct Acyclic Graph to provide a single temporal AST containing all versions at once. On top of the *HyperAST*, we take on the challenge of proposing novel AST matching algorithms, inspired by *GumTree*, on this DAG representation instead of the classical and inefficient analyzes of a pair of full ASTs. Essentially, we make the original greedy *GumTree* algorithms lazy. Our approach pre-computes metadata and lazily decompresses the DAG to decorrelates the cost of diffing from the size of the code base. Thus, enabling code differencing at scale.

We evaluated our approach on a curated list of 19 large software projects. Compared to *GumTree* as a baseline, the proposed approach outperforms it in scalability both in time and memory. We observed an order-of-magnitude difference in CPU time: 1) from $\times 1.2$ to $\times 12.7$ for the total time of diff computation, 2) from $\times 1$ to $\times 226$ for the top-down and bottom-up phases total time, and 3) from $\times 3.2$ to $\times 233$ for the top-down phase total time. We also observed an order-of-magnitude difference in memory footprint of $\times 4.5$ per AST node. Finally, we gain all the

time while having a 99.3% of identical diffs with respect to *GumTree* and 99.99975% of identical mappings and actions in the remaining 0.7% diffs. Finally, we also outperform *GumTree-Spoon* when including the parsing cost. We are faster by 14.52 times in median and when excluding extreme cases of gains, we are faster on average by 13.68 times.

This chapter follows the *Engineering Research Method* as we propose a novel scalable code differencing approach and evaluated through case studies supplemented with a replication package.

This chapter presents the following contributions:

— A novel approach for diffing commits that scales for the analyzes of large code histories.

— An evaluation of the proposal composed of benchmarking studies that demonstrates the ability of the approach to scale compared to a state-of-the-art code differencing approach.

— Open Science: All the code of the approach and the artifacts of the evaluation are freely available as a replication package. https://anonymous.4open.science/r/FSE23-DC22/

The rest of the chapter is structured as follows: Section 8.1 presents the novel approach. Section 8.2 details the evaluation and threats to validity. Section 8.4 concludes the chapter.

The approach and results presented this chapter were published at FSE 2023 [Le +23].

8.1 Contribution

Considering a code history represented using the *HyperAST*, we introduce a lazy code differencing approach inspired by the *GumTree* approach [Fal+14] with a faster and more efficient mapping of pairs of trees (that characterize commits). The contribution addresses the scalability issues as the following:
1/ the use of the HyperAST data structure overcomes the wasting of resources by not reusing intermediate computation results that could be shared among versions;
2/ the proposed lazy algorithm fixes the unadapted memory layout of compared trees by allocating nodes in the global heap;
3/ combining the HyperAST with the lazy diff algorithm reduces unneeded memory accesses to the ASTs during the diff algorithm.

The scaling capabilities of our approach benefit from the same hypothesis as the *HyperAST*: given a large code base, the amount of changes (*i.e.,* new subtrees) brought by each commit is usually tiny compared to the size of the code base. While both *HyperAST* and *GumTree* handle subtrees, *HyperAST* de-duplicates identical subtrees (*i.e.,* storing structural identical subtrees once). Thus, it might provide benefits when combined with specific algorithms for matching identical subtrees (that prune further traversal when matched). Compared to the *GumTree* approach, we propose to leverage the structure of the *HyperAST* to significantly reduce memory accesses and cache misses, while speeding up the diff.

Figure 8.1 shows the main differences between the *GumTree* approach and ours: The *GumTree* approach starts by parsing whole code files to then process all their resulting trees to compute metadata and finally produce diffs. It follows a top-down and bottom-up phases to compute the mappings between the original and evolved versions of the ASTs. After that, it computes the

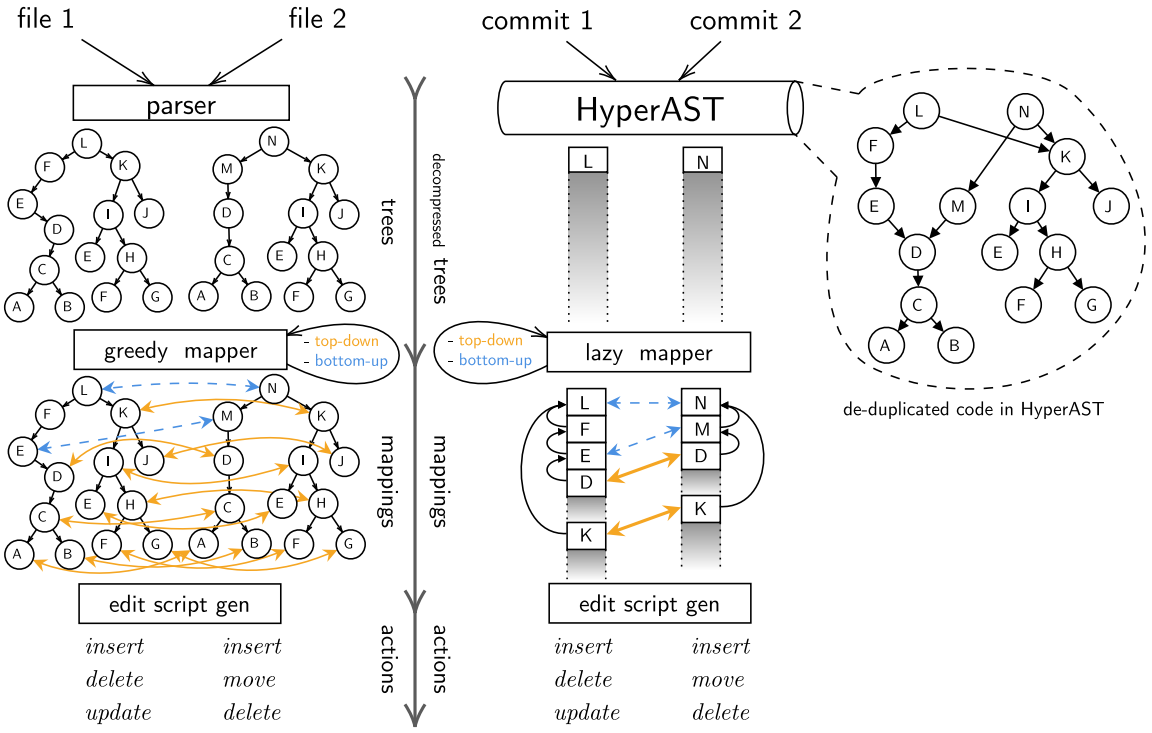


Figure 8.1 – *GumTree* pipeline (left) ; our pipeline with content of *HyperAST* (right)

diff. Instead, our approach relies on the *HyperAST* to efficiently process a code history, *i.e.,* a Git repository. For each version (*i.e.,* commit), our approach incrementally parses and computes metadata to persist. In addition to persisting metadata, the *HyperAST* also precomputes the structural equality using a reference equality, since identical subtree are de-duplicated. Then, pairs of trees from the *HyperAST* are lazily decompressed and mapped also in the same two phases (top-down and bottom-up) and finally used to produce diffs. Thus, the lazy decompression allows us to only focus on the changed parts of the code for diffing.

In this section, we first detail how the approach leverages *HyperAST* to provide structured code with metadata, and then how those structured data fit the requirements of each matching algorithm we propose. We then present the two specific matching algorithm, namely the lazy top-down matching and the lazy bottom-up matching. Our contribution herein is to adapt the original greedy algorithms of *GumTree* for the decompressed trees, leveraging on the lazy decompression. While optimizing the performances, our approach produces the same results as the original algorithms.

8.1.1 *GumTree* to *HyperAST* Metadata

To efficiently compare code elements, *GumTree* uses several precomputed metadata (see chapter introduction and section 6.1.3). As our approach leverages on the *HyperAST* that needs to compute and expose these metadata. The size and a hash (similar to the label hash) were already present in the *HyperAST*. Thus, we only extended the construction of the *HyperAST* (*i.e.,* parsing commits) to compute the label and structural hashes. In addition to these metadata, *GumTree* needs to test whether two subtrees are identical, *i.e.,* an *isomorphic* function. However,

due to the DAG nature of the *HyperAST*, identical subtrees are de-duplicated. Thus, we know that referentially identical subtrees are isomorphic without having to recursively compare their content as *GumTree* does in its implementation.

8.1.2 Obtaining a Tree from the *HyperAST*

We now present the compressed tree and its lazy decompression.

Decompressed tree

The *HyperAST* is a DAG where subtrees are unaware of their parents, as such, algorithms requiring global information on nodes (*i.e.,* subtrees) need additional structures. Global information refers to any information on parents of a node. It can be the global position of a node, its path, declaring class, file or offset in characters. In the remainder of this paper – as opposed to the compressed tree (the DAG) in *HyperAST*– we call such a structure that holds global information on nodes a **decompressed tree**. The process of extracting a decompressed tree from the *HyperAST* is named a **decompression**. To exploit spacial locality, a decompressed tree is represented by a contiguous array using a post-order layout. Actually, the bottom-up step mainly traverses trees in post-order, process subtrees (descendants) and other post-order properties, such as contiguous descendants, key roots and leftmost tree descendants. It has almost no downsides, other than having a $O(n)$ access time to access the n-th children (precomputing leftmost tree descendants is mandatory to obtain this complexity).

A decompressed tree has the following structure (inspired by the Zs algorithm [ZS89], see Section 2.1.3):

ids: an array of Ids that indexes subtrees in the *HyperAST*

llds: an array of integers that indexes leftmost tree descendants

parents: an array of integers that indexes parents

The decompressed tree is column-oriented, *i.e.,* is a struct of arrays,[1] while nodes are indexed by their position in post-order. In addition, considering that a decompressed tree is contiguous we are able to replace the uses of hash sets by bit sets.[2] Indeed, the original *GumTree* algorithm [Fal+14] uses existential quantifier ($\exists$) in various places and implemented by hash sets.

The downside of this decompressed tree would lie in the upfront cost of decompressing two entire versions before being able to compare them. Nonetheless, it is countered by the fact that it is lazily decompressed.

Lazy decompression

The upfront decompression of the *HyperAST* is actually not mandatory. Reducing and deferring the decompression effectively makes the decompression lazy. Indeed, considering the

[1] Structs of arrays (SoA) reduce memory wasted by padding, while helping with cache misses when only a subset of fields is needed.

[2] Using a bit sets to implement a set, is more memory efficient than a hash set (a single bit per element), considering modern virtual memory, where zeroed pages are not physically allocated.

Data: A source tree T_1 and a destination tree T_2, an multimap $\mathcal{MM}$, an empty list $\mathcal{A}$ of candidate mappings, and an empty set of mappings $\mathcal{M}$, the minimum height for a matched subtree $minHeight$
Result: The set of mappings $\mathcal{M}$

$L_1 \leftarrow PList(root(T_1))$; $L_2 \leftarrow PList(root(T_2))$;
while $min(peekMax(L_1), peekMax(L_2)) > minHeight$ **do**
  if $peekMax(L_1) \neq peekMax(L_2)$ **then**
    if $peekMax(L_1) > peekMax(L_2)$ **then**
      foreach $t \in$ `pop(`L_1`)` **do** `open(`t`,`L_1`)`;
    else
      foreach $t \in$ `pop(`L_2`)` **do** `open(`t`,`L_2`)`;
  else
    $H_1 \leftarrow$`pop(`L_1`)`; $H_2 \leftarrow$ `pop(`L_2`)`;
    $b_1 \leftarrow \underline{BitSet(|H_1|)}$; $b_2 \leftarrow \underline{BitSet(|H_2|)}$;
    foreach $(i_1, i_2) \in 0..|H_1| \times 0..|H_2|$ **do**
      if $\underline{isomorphic(H_1[i_1], H_2[i_2])}$ **then**
        $\underline{link(\mathcal{MM}, H_1[i_1], H_2[i_2])}$;
        $b_1[i_1] \leftarrow 1$; $b_2[i_2] \leftarrow 1$;
    foreach $i_1 \in 0..|H_1|$ ***if*** $\neg b_1[i_1]$ **do** `open(`$H_1[i_1]$`,` L_1`)`;
    foreach $i_2 \in 0..|H_2|$ ***if*** $\neg b_2[i_2]$ **do** `open(`$H_2[i_2]$`,` L_2`)`;
$ignored \leftarrow \underline{BitSet(|T_1|)}$;
foreach $t_1 \in allSrcs(\mathcal{MM})$ **do**
  $uniq \leftarrow \bot$;
  if $|dsts(\mathcal{MM}, t_1)| == 1$ **then**
    $t_2 \leftarrow dsts(\mathcal{MM}, t_1)[0]$;
    if $|srcs(\mathcal{MM}, t_2)| == 1$ **then**
      $uniq \leftarrow \top$;
      add all pairs of isomorphic nodes of $\underline{s(t_1)}$ and $\underline{s(t_2)}$ to $\mathcal{M}$;
  if $ignored[t_1] \vee uniq$ **then** continue;
  foreach $t_1 \in srcs(\mathcal{MM}, dsts(\mathcal{MM}, t_1)[0])$ **do**
    $ignored[t_1] \leftarrow 1$;
    foreach $t_2 \in dsts(\mathcal{MM}, t_1)$ **do**
      $add(\mathcal{A}, (t_1, t_2))$;
$sort\ (t_1, t_2) \in \mathcal{A}$ using $\underline{sim(t_1, t_2, \mathcal{M})}$
$ignored\_src \leftarrow \underline{BitSet(|T_1|)}$; $ignored\_dst \leftarrow \underline{BitSet(|T_2|)}$;
foreach $(t_1, t_2) \in \mathcal{A}$ **do**
  if $\neg ignored\_src[t_1] \wedge \neg ignored\_dst[t_2]$ **then**
    add all pairs of isomorphic nodes of $\underline{s(t_1)}$ and $\underline{s(t_2)}$ to $\mathcal{M}$;
    $ignored\_src[t_1] \leftarrow 1$; $ignored\_dst[t_2] \leftarrow 1$;
    foreach $t \in s(t_1)$ **do** $ignored\_src[t] \leftarrow 1$;
    foreach $t \in s(t_2)$ **do** $ignored\_dst[t] \leftarrow 1$;

Algorithm 2: Lazy subtree matching

original *GumTree* matching algorithms and depending on the amount of changes and where they are located, entire subtrees might remain unchanged and will be matched early (using metadata and referential equality) without ever needing to access their descendants. In those cases, there is no need to decompress these subtrees. In Figure 8.1, a subtree with compressed descendants is materialized by dotted cells (fat red arrows means all descendants are matched uniformly). To control the decompression process while computing the diff, we provide three different methods to decompress a tree T:

***decompress_children*(T, t):** decompresses in T the children of the node located at position t.

***decompress_to*(T, t):** decompresses in T the node located at position t. It also decompresses all its parents with the method *decompress_children*.

***decompress_descendants*(T, t):** decompresses in T the descendants located at position t. It offers optimization opportunities when considering the layout of the decompressed tree (*e.g.,* post-order). Indeed, for the post-order layout, it is possible to reduce the number of accesses to the *HyperAST* with a stack that allows to defer the insertion of children when decompressing a node. Thus, there is no need to access the size (metadata in the *HyperAST*) of those children before actually decompressing them.

Each decompression method is incremental. Thus, making the overall decompression incremental. To check whether a node is decompressed, we check if its parent is 0 (*i.e.,* its initial value). This property always holds, except for the edge case where $|T| \leq 1$, *i.e.,* the tree is a single node. These three methods are used as follows in the proposed matching algorithms: Methods *decompress_children* and *decompress_descendants* replace every call to the original (non decompressing) children accessor. The method *decompress_descendants* is specifically used when most or all descendants need to be decompressed. The method *decompress_to* should be used when a specific node needs to be decompressed.

8.1.3 Lazyfied Top-down Mapping Phase

Algorithm 2 [3] summarizes the top-down phase that matches the largest isomorphic subtrees between the source T_1 and target T_2. The underlined expressions in Algorithm 2 represents our optimizations for lazifying the *GumTree* top-down phase.

The following constructs are required to understand Algorithm 2:

— $root(T)$ is the root node of T.

— $s(t)$ returns the list of descendants in post-order, *i.e.,* from $lld(t)$ to t (see Section 8.1.2).

— $BitSet(n)$ is an array of bits of size n.

— $PList(t)$ creates a height-indexed priority tree list L containing the subtree t.

— $peekMax(L)$ returns the greatest height of the list.

— $pop(L)$ takes the list of greatest height subtrees from L.

[3] Note that we use in Algorithms 2 to 4 the same hyperparameters as *GumTree* [Fal+14], namely, $minHeight = 2$, $maxSize = 100$, and $minDice = 0.5$.

— $open(L, t)$ inserts all the children of t into L. Algorithm 3 presents this function. Compared to *GumTree* we changed the access to children into a decompression of said children (using $decompress_{children}$). Then, instead of accessing a precomputed height directly on the node (as *GumTree* does), we need to access the subtree corresponding to *HyperAST*, first recovering the identifier (with *original*, see Section 8.1.2) to the *HyperAST*, then accessing the height metadata (with *height*, see introduction of chapter 8)

— $sim(t_1, t_2)$ computes a similarity distance between t_1 and t_2. It ranges from 0 to 1, where a value of 1 indicates that the descendants of T_1 are the same as those of T_2.

Data: a subtree $t \in T$, T being layouted in post-order, and a height-indexed priority list L containing subtrees of T
Result: The range of descendants in post-order

```
1 foreach t' ∈ decompress_children(t) do
2     h ← height(original(t')) − 1;
3     if h > minHeight then
4         level ← maxHeight − h;
5         L[level] += t;
```

Algorithm 3: Open a subtree in priority list

$\mathcal{MM}$ represents a structure containing multi-mappings, *i.e.,* mappings where nodes can be part of multiple mappings as opposed to $\mathcal{M}$ that only contains distinct mappings.

— $allSrcs(\mathcal{MM})$ returns all mapped sources in $\mathcal{MM}$.

— $srcs(\mathcal{MM}, t_2)$ returns all source nodes mapped to t_2 in $\mathcal{MM}$.

— $dsts(\mathcal{MM}, t_1)$ returns all destination nodes mapped to t_1 in $\mathcal{MM}$.

— $link(\mathcal{MM}, t_1, t_2)$ maps t_1 and t_2 in $\mathcal{MM}$.

Algorithm 2 follows three steps. The first step (lines 1-20) calculates the multi-mappings (MM) between the largest isomorphic subtrees. It maps isomorphic subtrees (Lines 15-16), while iteratively opening unmapped subtrees. More specifically, when the heights are not equals for the subtrees (Lines 4-8) or when they are not mapped in Line 16 with $b_i = 0$ (Lines 19-20).

The second step of Algorithm 2 (Lines 21-34) moves multi-mappings stored in $\mathcal{MM}$ to a list of mappings $\mathcal{A}$ (Line 34), while directly moving mono-mappings (*i.e.,* mappings between exactly 2 nodes) to $\mathcal{M}$ (Line 28). It mainly serves as a preprocessing phase before sorting and filtering multi-mappings. It also removes mono-mappings, and hence, reduces the number of mapping to sort and filter. It uses a bit set to mark nodes that are already mapped (Line 21). Then, Algorithm 2 sorts the list mappings in $\mathcal{A}$ using a similarity distance (Line 35), such as the *dice* distance used in *GumTree* [Fal+14].

The last step (Lines 36-44) extracts mappings from $\mathcal{A}$ to $\mathcal{M}$ (Line 40), while filtering overlapping mappings with an already accepted one, *i.e.,* sharing nodes. Nodes accepted in $\mathcal{M}$ are marked by *ignored_src* and *ignored_dst* (Line 41-44) that forbids them from being accepted later (Line 39).

8.1.4 Lazyfied Bottom-up Mapping Phase

Data: A source tree T_1 and a destination tree T_2, and an empty set of mappings $\mathcal{M}$, a threshold *minDice* and a maximum tree size *maxSize*
Result: The set of mappings $\mathcal{M}$

```
1  foreach s1 ∈ T1 | s1 is not matched, in post-order do
2      t1 ← decompress_to(T1, s1);
3      if t1 has no matched children then continue;
4      t2 ← candidate(t1, M);
5      if t2 ≠ null ∧ dice(t1, t2, M) > minDice then
6          M ← M ∪ (t1, t2);
7          if |s(t1)| < maxSize ∨ |s(t2)| < maxSize then
8              u1 ← decompress_descendants(t1);
9              u2 ← decompress_descendants(t2);
10             R ← opt(u1, u2);
11             foreach (ta, tb) ∈ R do
12                 if ta, tb not already mapped ∧ type(ta) = type(tb) then
13                     M ← M ∪ (ta, tb);
```

Algorithm 4: Lazy bottom-up matching

The bottom-up phase, as shown in Algorithm 4,[3] complements the top-down phase, leveraging the previously mapped subtrees in $\mathcal{M}$ to further map the remaining nodes. The underlined expressions represent our optimizations for lazifying the *GumTree* bottom-up phase. Using the subtrees mapped in previous phase, Algorithm 4 is able to map slightly different nodes (*i.e.,* not isomorphic nodes). The matcher first compares the number of shared descendants, to then match subtrees smaller than *minHeight* (leveraging existing optimal mapping algorithms).

With the bottom-up phase, in post-order, we aim to map remaining (unmapped) source nodes (Algorithm 4) to destination nodes. We only decompress the unmapped source nodes (Algorithm 4) before skipping the nodes with no matched children (Algorithm 4). Then, the auxiliary *candidate* function is used to find a candidate destination node t_2 (Algorithm 4) most similar to t_1. Using the Dice distance (compared to *minDice GumTree* parameter), if t_1 and t_2 are similar enough (Line 5), they are matched in $\mathcal{M}$ (Algorithm 4). If t_1 and t_2 have a small number of descendants (compared to *maxSize GumTree* parameter), their descendants are then decompressed and provided to *opt* (Lines 7-10). *opt* is an optimal matching algorithm (such as Zs) that matches nodes of u_1 and u_2 while minimizing the edit distance. Finally, only mappings with unmatched nodes and same types are kept and added in $\mathcal{M}$. To generate the diff, we use the same algorithm of Chawathe *et al.* [Cha+96] without lazifying it in this paper.

8.2 Evaluation

This section presents the evaluation of our code differencing approach. First, we present the research questions. We then present the data set and evaluation process. After that, we

present our evaluation protocol and the obtained results. We finally discuss the threats to validity, limitations, and the scope of the approach. **All the material of this section and a replication package are available on our companion web page**.[4] We ran the implementation of our approach on the following hardware configuration: *2 x Intel(R) Xeon(R) Gold 6238 CPU @ 2.10GHz; 187Gb ram; 1 T SSD*, running *Ubuntu 18.04.6*.

8.2.1 Research Questions

We formulate the research questions as follows:

RQ1 **To what extent can our approach produce identical results as the state-of-the-art technique?** This aims to investigate the soundness of the produced mappings and diff compared to a ground truth, namely GumTree.

RQ2 **To what extent does our approach perform and scale on the memory footprint of computing diffs compared to a state-of-the-art approach?** This aims to position the scalability performance on memory consumption of our diffing algorithm over a long evolution history with an established state-of-the-art solution. In particular, we measure the memory heap allocated per node.

RQ3 **To what extent does our approach perform and scale on the time performance of computing diffs compared to a state-of-the-art approach?** This aims to position the scalability performance on execution time of our diffing algorithm over a long evolution history with an established state-of-the-art solution. In particular, we measure the total time to compute a diff and the time for the two phases of top-down and bottom-up of the mappings.

RQ4 **To what extent does our approach perform compared to a state-of-the-art approach on a practical use case of parsing and diffing commits?** The previous RQs evaluated the performance of our commit diffing algorithm. Thus, ignoring the AST preparation (extraction and construction) from a history. RQ4 measuring the cost of parsing the commits along with the cost of computing the diffs. It does on a concrete use case as experienced by a developer that computes commit diffs on a code history.

8.2.2 Dataset

Table 8.1 details the final list of software projects we used in the following evaluation. The evaluation re-used the dataset employed in the *HyperAST* paper [Le +22] as it has been already peer-validated and it matches our following requirements:

— Large open-source real-world *Java* projects to work on representative code;

— A sizable number of contributors per project to work on commits made by different developers;

— A large number of commits per history to work on representative code histories;

— *Java* 14 support, since GumTree 3.0 works with JDT 3.26.

[4] https://anonymous.4open.science/r/FSE23-DC22/

Table 8.1 – Data set characteristics

Projects	# LoC	# files	Commits	Contributors	Stars
Apache Hadoop	1.63M	10.2k	25,749	435	12k
Apache Flink	1.5M	13.2k	30,587	1,037	1.8k
Quarkus	614k	10.5k	29,635	616	9.7k
Google Guava	509k	3.16k	5,794	273	44k
Netty	317k	2.78k	10,789	569	29k
Apache Dubbo	197k	2.81k	5,437	393	37k
Alibaba fastjson	188k	3.12k	3,946	176	24k
Apache Log4j2	183k	2.32k	12,031	132	2.8k
Jenkins	181k	1.69k	32,252	701	19k
Javaparser	179k	1.67k	8,031	166	4.1k
Inria Spoon	154k	2.06k	3,891	106	1.3k
AWS Toolkit Eclipse	93.9k	1.08k	111	21	27k
Apache Maven	92.5k	1.05k	11,567	150	3.1k
Apache Spark	85.6k	1.06k	32,821	1,805	33k
Apache SkyWalking	84.7k	1.58k	7,022	397	1.9k
Jackson Core	52.3k	283	2,025	59	200
Alibaba Arthas	44.2k	586	1,726	155	29k
Google gson	25.8k	212	1,650	124	21k
SLF4J	13.5k	256	1,956	61	1.9k

8.2.3 Evaluation Protocol

We now present the experimental protocol we followed for the evaluation. As *GumTree* is the most advanced state-of-the-art differencing tool, we select it as a baseline. The evaluation protocol is divided into three parts: one protocol for RQ1, another one for RQ2 and RQ3, and one specific protocol for RQ4. These four RQs use several of the following five objects:

GumTree. The original version of *GumTree* in *Java* without its *Java* parser. Used in RQ1 to RQ3.

Lazy. The current proposed approach (in Rust), relying on the *HyperAST* version with lazy top-down and bottom-up phases. Used in RQ1 to RQ4.

Not lazy. The closest equivalent of *GumTree Java* but in Rust and relying on the *HyperAST*. This object is useful to mitigate comparison between *Java* and *Rust* program executions. This version still benefits from the *HyperAST* but no lazy phases. Used in RQ3.

Partial lazy. Similarly to *Not lazy* but lazy on the top-down phase. Useful to measure the effect of not lazifying during the bottom-up phase. Used in RQ3.

GumTree-Spoon.[5] The original version of *GumTree* in *Java* backed with its official *Java* parser (*Spoon*). Used in RQ4.

We now detail the protocol for each RQ:

⇒ **RQ1.** It compares the results of our approach (object *Lazy*) to the baseline (*i.e.,* object *GumTree*). To do so, we check whether each diff *Lazy* and *GumTree* produced are identical. The

[5]https://github.com/SpoonLabs/gumtree-spoon-ast-diff

independent variables are the mappings found in a diff and the actions a diff contains. Thus, We compare the mappings and the diff's actions for our approach against GumTree.
⇒ **RQ2.** It focuses on the following two objects: *GumTree*, *Lazy*. *Partial Lazy* and *Not Lazy* are not discussed here because they only differ from *Lazy* about when memory is used, not the total amount consumed. The independent variables for RQ2 are: the measured memory heap allocated using the objects *Lazy* and *GumTree*; the number of nodes used in both objects (both *Lazy* and *GumTree* use the same number of nodes). We then divided the measured heap size by the number of nodes to obtain a result in byte per node.
⇒ **RQ3.** It studies the four following objects: *GumTree*, *Lazy*, *Not Lazy*, *Partial Lazy*. The independent variable in RQ3 is the execution time. First the total time spent: top-down, bottom-up and computing the diff actions from the mappings with Chawathe algorithm [Cha+96]; then the two matching phases: top-down and bottom-up; finally only the top-down phase.
RQ1-3. Regarding RQ1 to RQ3, the first step of the protocol consists in parsing the various commits to provide to *GumTree* and to construct the *HyperAST*. This is a preprocessing step before computing the diffs. Then, we provide the resulting parsed trees to our approach and to GumTree. To precisely evaluate the commit diffing algorithm, we provide *HyperAST* and *GumTree* with the same ASTs for these three RQs. Only after that, we start measuring the performance of the different phases and algorithms for computing the diffs. So, the parsing time for *GumTree* and construction time of the *HyperAST* are not considered in the evaluation in the RQ1, RQ2, and RQ3. Thus, ensuring a more controlled measurements of the algorithmic performances and an unbiased comparison.
⇒ **RQ4.** It studies the two following objects: GumTree-Spoon and *Lazy*. The independent variable in RQ4 is the execution time. We compare the spent time at computing diffs of the latest 100 commits for each project while including the ASTs parsing time for *GumTree-Spoon* and construction time of the *HyperAST*. Indeed, the combination of Spoon with *GumTree* allow us to feed entire commits as ASTs to the *GumTree* algorithm with the Spoon parser.

8.2.4 Results

We now present and discuss the observed results.

RQ1: To what extent can our approach produce identical results as the state-of-the-art technique?

To answer RQ1, we compare the mappings and diffs produced by our approach to the *GumTree* baseline. In total, we calculated 18 092 diffs implying 919 132 mappings.[6] 17 972 (99.3%) of these diffs were identical by matching the *GumTree* results identically. 99.999% of the 919 132 mappings of the diffs are also identical.

We manually scrutinized and checked the other 120 (0.7%) diff and found out the following edge cases. 1) 64 cases were *GumTree* had a diff generation error, and hence, not computing

[6]Distribution of change size per diff is given in our anonymous github in size-plot.png. Varying from small changes, to very large changes several commits.

fully the final diff actions. However, when comparing their mappings, they were identical. 2) 2 cases where our diff had 2 more actions than the *GumTree* diff. 3) 54 other cases where our diff had less actions than the *GumTree* diff. This last 54 cases are explained by the fact that we could calculate more mappings between the AST nodes than what *GumTree* did. In fact, these cases highlight better diffs since they do not contain additional add and delete actions due to unmapped AST nodes. Overall, even in these 120 diffs, 99.999% of mappings and 99.943% of the diff actions are identical. Only, few mappings and diff actions cause comparison issue. Therefore, we consider these marginal cases as outliers due to implementation issues in our prototype or our execution environment in comparison to the more than 99% of correct diffs and mappings.

***RQ*1 insights:** The results show that 99.3% of the diffs and 99.999% of the mappings our approach produced are identical to the *GumTree* outputs. 120 diffs where not identical to GumTree. Still, they remain similar at 99.943% of diff actions and at 99.999% of mappings. Thus, our approach produces identical results that *GumTree* on the involved data set.

RQ2: To what extent does our approach perform and scale on the memory footprint of computing diffs compared to a state-of-the-art approach?

To answer RQ2, we measure the memory heap allocated given the same number of nodes for our approach and for GumTree. On average, *GumTree* needs 74.4 bytes per node. Our implementation needs 0.278 byte per node in the DAG (over 1000 commits) and 16.13 additional bytes per decompressed node. Moreover, considering that our approach decompresses nodes lazily and the allocation of a zeroed (0) and contiguous piece of memory, on modern operating systems such an allocation is deferred at the granularity of a virtual memory page. Thus, with our approach physical memory is only allocated when a node is decompressed (writing something not trivially zero). Yet, the peak (transient) memory footprint of our approach occurs during the diff generation at the same stage as the original*GumTree* implementation. Indeed, we did not make the Chawathe *et al.* [Cha+96] algorithm lazy as it is not a focus of the core contribution on computing the mappings, both in our approach and GumTree. In addition, the Chawathe algorithm (diff generation) other memory constraints as it uses a third tree that starts as a copy of the source tree and is mutated for each change until it structurally becomes the destination tree.

***RQ*2 insights:** Results show out low memory footprint compared to *GumTree* with, on average, 0.278 + 16.13 bytes per node in our approach versus 74.4 bytes per node in GumTree. Representing an order-of-magnitude difference of $\times 4.5$.

RQ3: To what extent does our approach perform and scale on the time performance of computing diffs compared to a state-of-the-art approach?

Herein, we measure time at the three different stages of the *GumTree* algorithm, namely the total time spent; the two matching phases (top-down and bottom-up phases); and the top-down phase only. Figures 8.2 to 8.4 shows respectively the results for the three aforementioned stages.

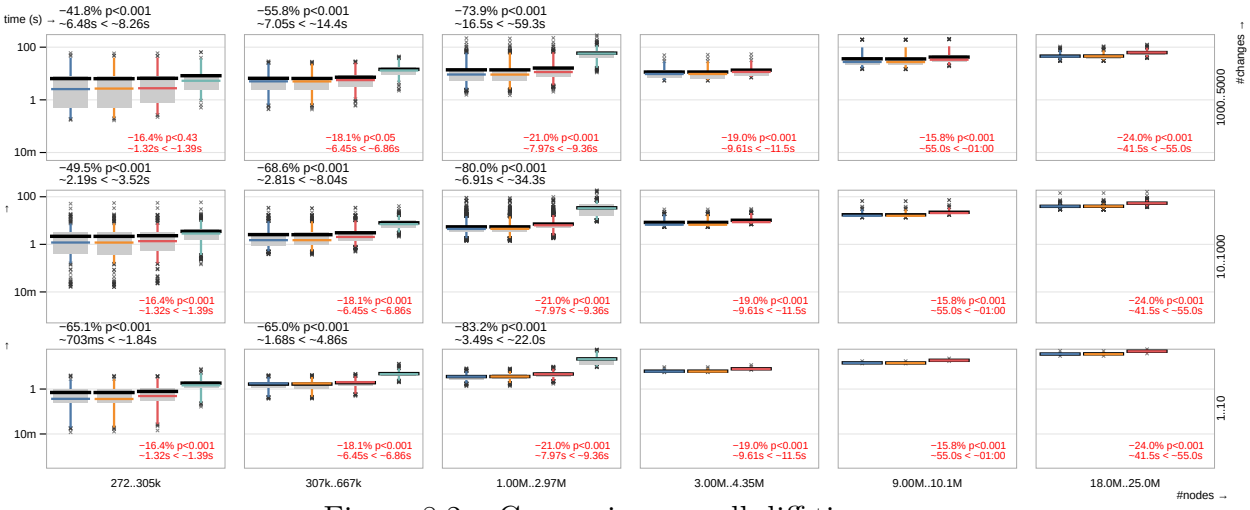


Figure 8.2 – Comparing overall diff time;
Color Legend: [lazy: blue, partial lazy: orange, not lazy: red, GumTree: green]
Facet Legend 1 in black: [relative gain in %, p-value (Mann-Whitney); avg. for our approach, avg. for GumTree]
Facet Legend 2 in red: [relative gain in %, p-value (Mann-Whitney); avg. for our lazy approach, avg. for our non-lazy variant]

Each figure uses the same plotting scheme: the time taken is displayed as vertical box plots on a logarithmic scale where the *mean* is a thick horizontal black line and the *median* is a thin colored horizontal line. The box delimits the first and third quantile. The vertical bar delimits the 95% confidence interval. Extreme points are displayed outside this interval.

Figures 8.2 to 8.4 shows groups of four box plots corresponding to the four objects we compare (see Section 8.2.3), respectively: *Lazy* (our approach, in blue), *Partial lazy*, *Not lazy* (in red), and *GumTree* (in green). We faceted (*i.e.,* grouped) the figure in both axis, due to the correlation of computation time both with: 1) the size of the output (*i.e.,* the diff) horizontally with three groups (rows): 1 to 10, 10 to 1000, and 1000 to 5000 changes. The number of changes is the size of the diff in terms of number of modifications needed to transform one AST version to the other. 2) the size of each version vertically in terms of number of nodes, from hundred thousands to millions in 6 groups (columns). On top of each facet (*i.e.,* groups of box plots) in black, we put the relative gain in %, the p-value, and the average time for our approach and GumTree. In red, we display the same information for our lazy and the non-lazy variant.

Figure 8.2 presents the total time taken to compute the diff including the top-down and bottom-up mapping phases with the diff generation. We can first observe that our approach outperforms the original *GumTree* each time, gaining between 42% (avg. 6.48 s vs 8.26 s) and 83% (avg. 4.49 s vs 22 s) of reduced overall computation time. The gains are more important with a lower number of changes relative to the size of the codebase. In particular, Between 1 and 10 changes, our gains increase up to 83% of execution time compared to the baseline.. Then, between 10 and 1000 changes, our gains increase up to 80%. Finally, for more than 1000 changes our gains improve up to 74%. In addition, we observe that our approach with the lazy implementation also outperforms the non-lazy variant all the time, regardless of the size of the code and the size of the diff. The gain varies from 16% (avg. 55 s vs 60 s) to 24% (41.5 s vs 55 s).

Moreover, for the four largest projects with more than 300KLOC and 3M nodes, we disabled

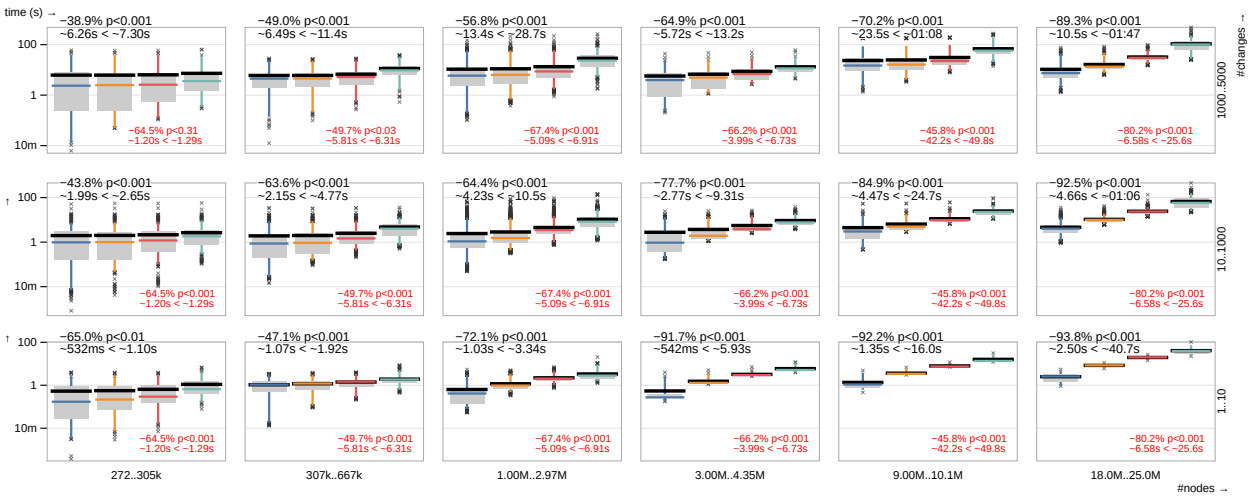


Figure 8.3 – Comparing the top-down and bottom-up phases time;
Color Legend: [lazy: blue, partial lazy: orange, not lazy: red, GumTree: green]
Facet Legend: [relative gain in %, p-value (Mann-Whitney) ; avg. for our approach, avg. for GumTree]
Facet Legend 2 in red: [relative gain in %, p-value (Mann-Whitney); avg. for our lazy approach, avg. for our non-lazy variant]

the generation of the diff for the *GumTree* implementation. Indeed, we observed extreme slowdowns of *GumTree* when enabled, leading almost every time to out of memory errors. As we did not attempt to lazify the generation of the diff, we further measured the time performance without the generation, *i.e.,* the top-down and bottom-up phases only and the time performance of the top-down phase only.

Figure 8.3 presents the time taken to compute the mappings during the top-down and bottom-up matching phases. Herein, our approach outperforms the original *GumTree* top-down and bottom-up phases each time, gaining between 39% (avg. 6.26 s vs 7.30 s) and 94% (avg. 2.50 s vs 40.7 s) of reduced overall computation time of the mappings. Between 1 and 10 changes, our gains increase from 47% to 94%. Then, between 10 and 1000 changes, our gains increase from 44% to 93%. Finally, for more than 1000 changes our gains improve from 39% to 90%. Similarly, our lazy variant outperforms the non-lazy variant all the time. The gain varies from 46% to 80%.

Figure 8.4 presents the time taken to compute only the mappings with the top-down matching phase. Herein, our lazy top-down phase outperforms the original *GumTree* top-down each time, gaining between 89% (avg. 91.1 ms vs 510 ms) and 98% (avg. 5.38 ms vs 235 ms) of reduced overall computation time of the top-down mappings. Between 1 and 10 changes our gains increase from 95% to 98%. Between 10 and 1000 changes, they increase from 95% to 96%. Finally, for more than 1000 changes they increase from 88% to 95%. Similarly, our lazy variant largely outperforms the non-lazy variant all the time. The gain varies from 87% to 94%.

These three figures highlight our lazy approach significantly gains during the top-down phase (in percentage) and the bottom-up phase (in absolute time). While our diff generation is faster than the *GumTree* one and scales up to thousands of actions on large software projects (*e.g.,* Hadoop), further gain could be achieved. Indeed, we did not lazify the diff generation, meaning that a full decompression of the trees is mandatory before generating the diff. Lazifying the generation is future work. It is worth noting that for the results of the lazy variant, a full decompression is done before computing the diff, taking 26.6% of the generation time while

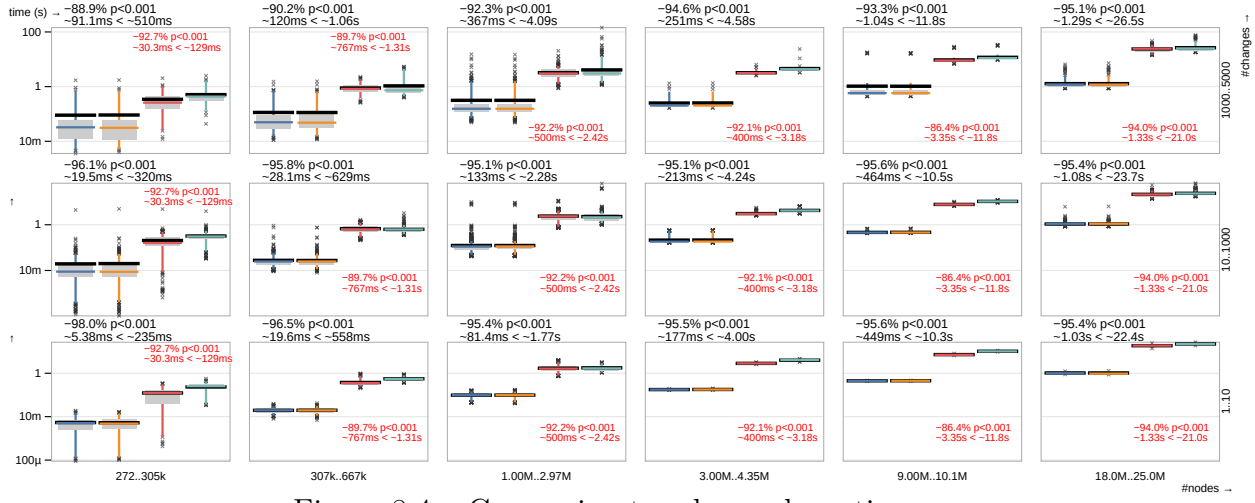


Figure 8.4 – Comparing top-down phase time;
Facet Legend: [relative gain in %, p-value (Mann-Whitney) ; avg. for our approach, avg. for GumTree]
Facet Legend 2 in red: [relative gain in %, p-value (Mann-Whitney); avg. for our lazy approach, avg. for our non-lazy variant]
Color Legend: [lazy: blue, partial lazy: orange, not lazy: red, GumTree: green]

only 12.0% for the non-lazy variant. Furthermore, in Figure 8.3 we observe that the bigger the projects, the better our lazy approach performs compared to the two variants *Partial lazy* and *Not lazy*. In Figure 8.4, we also observe the same gain compared to the *Not lazy* variant.

> ***RQ3* insights:** Results show a systematic and significant gains of 83% on average and up to 99% of our lazy approach compared to *GumTree* in all phases, from the top-down and bottom-up matching phases to the generation of the diff. Representing an order-of-magnitude difference in total time: 1) from ×1.2 to ×12.7 for diff computation, 2) from ×1 to ×226 for the top-down and bottom-up phases, and 3) from ×3.2 to ×233 for the top-down phase.

RQ4: To what extent does our approach perform compared to a state-of-the-art approach on a practical use case of parsing and diffing commits?

To answer this RQ, we measured the execution time for parsing and diffing 100 commits for each project. Hence, we can compare the overall performance in a practical use case similarly as a developer would go through to compute diffs on given number of commits. To do so, we use the GumTree-Spoon version that parses a commit and then calls the *GumTree* diffing algorithm.

Figure 8.5 depicts the time of our approach including the construction of the *HyperAST* in orange versus the computation time for GumTree-Spoon including the parsing of the commits. We observe that our approach outperforms GumTree-Spoon on all the projects. For our largest projects Hadoop and Flink, GumTree-Spoon crashes most of the time during the generation of the diffs, due to using more than 32 GB of memory heap. [7] We also had other cases of crashes in Jenkins and in Gson. These cases are in red rather than orange in fig. 8.5. In *SkyWalker*,

[7] with the JVM disabling this limit mendates to go from 32 bits pointers to 64 bits and poses other issues (even bigger footprint, cache misses).

GumTree-Spoon could not parse completely most of the commits due to an issue with the multi-modules of *Java*. GumTree-Spoon could only parse some modules, sometimes one module only. This explains why in many commits it was faster than our approach that did diff all modules. Overall, when computing diff successfully, our approach was on 14.52 times faster than GumTree-Spoon in half of the cases (median). We had extreme cases of improvement in the six last small projects (from Jackson to slf4j) where we were thousands and hundred thousands times faster than GumTree-Spoon. Excluding these projects, we reach an average of 13.68 times where we are faster.

We also investigated the case of diffing only two commits by looking at the first two commits in our projects, since we must construct the *HyperAST* entirely in the first commit and incrementally update it in the second commit, before computing the diff. We see that we could compute the diff faster than GumTree, as shown by *t[0]* in Figure 8.5 (note that it is in two formats min:sec or 0.millisec). For example, in maximum in Hadoop and Flink, our approach took, respectively 1 min 24 s and 1 min 10 s while *GumTree* took 21 min 57 s and 24 min 3 s. In minimum in Slf4j, we took 300 ms while *GumTree* took 36 s. In other medium-sized projects as netty, dubbo, log4j, jenkins, javaparser, spoon, maven and spark, we respectively took from 3 s to 27 s, while *GumTree* took from 19 s to 3 min 35 s. On the smallest four projects, our approach varied from 300 ms (0.3 s) to 675 ms, whereas GumTree-Spoon varied from 9 s to 36 s. Therefore, even on two commits where we must build the *HyperAST* from scratch, a developer benefits of our lazified approach based to diff two commits or more. We could reach a gain up to 99% and an order-of-magnitude of ×122 when considering the two first commits only.

***RQ4* insights:** Our lazy approach outperforms GumTree-Spoon. We are faster by 14.52 times in half of the cases (median) and when excluding extreme cases of gains, we are faster on average by 13.86 times. We also outperform GumTree-Spoon on the basic use case of diffing two commits only, with a gain up to 99% and an order-of-magnitude of ×122.

8.2.5 Threats to Validity

This sections discusses threats to validity w.r.t. [Woh+12].

Internal Validity

Considering the computation of diffs, we first had to evaluate its ability to produce identical outputs (mappings and diffs) as GumTree. To make sure we have unbiased measurements, we used the *HyperAST* to construct the code history and we retrieved the trees of each commit from the *HyperAST*. Thus, we had a uniform representation of the node elements (*i.e.,* same parser and same grammar for the ASTs) for comparing our approach and GumTree. Moreover, our implementation and the *HyperAST* are implemented in Rust while *GumTree* is developed in *Java*. To mitigate this difference of language while comparing execution time and memory usage, we provide and compared our approach with two other objects developed in Rust using the *HyperAST*: *Partial lazy* and *Not lazy*. *Not lazy* is the closest Rust version of *GumTree* while

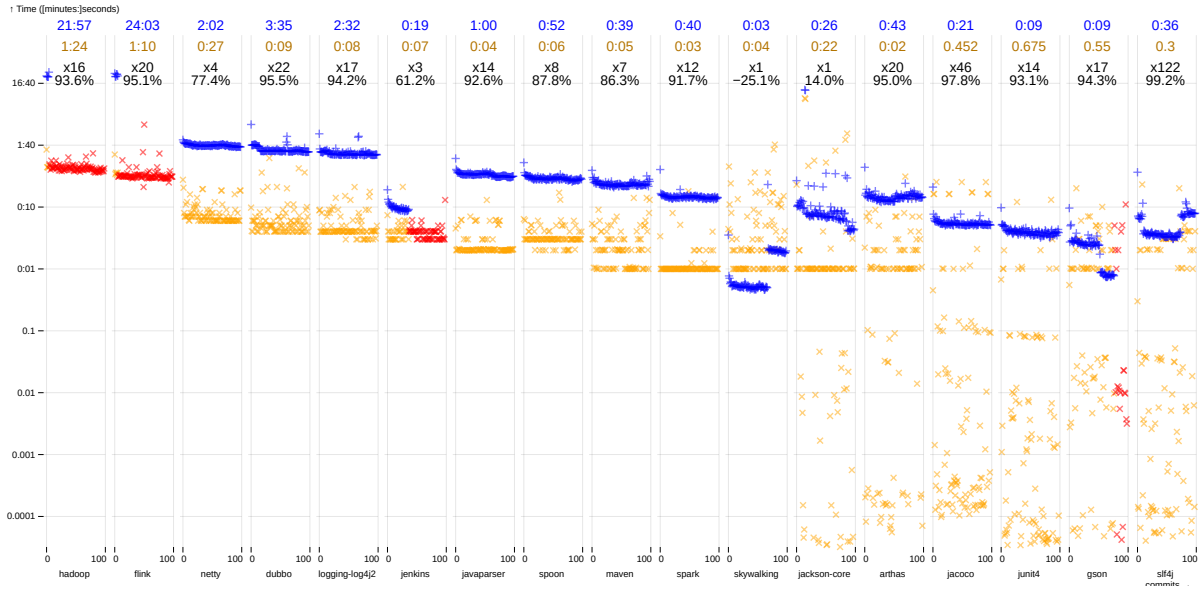


Figure 8.5 – Comparing overall execution time (parsing + diff) on 100 commits;
Facet Legend: [*t[0]* time to compute first diff between first and second commit, lazy, *GumTree*, format min:sec or 0.millisec]
Color Legend: [lazy: orange times, lazy while GumTree-Spoon failed: red times, GumTree-Spoon: blue plus]

still benefiting of the *HyperAST*. Compared to these two variant objects, our evaluation still shows significant benefits for our approach with all lazy phases.

External Validity

The evaluation implied 19 projects. We carefully follow a clear protocol to select relevant and significant *Java* projects. Our curated list of projects represents real-world complex software systems with very large histories.

We implemented our approach on top of the *HyperAST* by lazifying the *GumTree* algorithms. We then evaluated our approach on projects written mainly in *Java* and build with *Maven*. Our conclusions in theory could generalize to other programming languages with similar features as Java (*e.g.,* strong static nominal typing). Nonetheless, further experimentation remains necessary on other languages to generalize our results. We also cannot generalize the diffing results to other diffing algorithms, such as [Flu+07]. Note that as the *HyperAST* supports only Java so far, we could only evaluated and compared to *GumTree* on Java projects. However, the goal of this paper was not to support multiple languages but to show the scalability of computing diffs on large code history. Besides, *GumTree* performance are not language dependent [Fal+14].

Construct Validity

Our evaluation shows that our approach scales the computation of diffs on large code history and large projects representing real-world complex software with very large histories. Compared to GumTree, we outperformed it by an order-of-magnitude difference in CPU time from ×1.2 to ×12.7 for the total time of diff computation and up to ×226 in intermediate phases of the diff

computation, and an order-of-magnitude difference in memory footprint of ×4.5 per AST node. Further evaluation remains necessary for more insights and statistical evidence.

8.3 Data Availability

All the materials used to evaluate HyperDiff and a replication package are available at: https://gihub.com/quentinLeDilavrec/FSE23/
It contains:

— a snapshot of our implementation,

— the notebooks used to compute results and plots for the evaluation, and

— the snapshot of baseline (GumTree) along a generator that enable to use the exact same AST topology, utils to make comparison of results faster, and some instrumentation to prescisely measure performances.

The implementation and benchmark helpers are part of our mono-repo:
https://gihub.com/HyperAST/HyperAST/

8.4 Conclusion

This chapter introduced a novel code differencing approach that scales to large code histories composed of millions of nodes, leveraging the *HyperAST* representation of code histories to further lazify the GumTree algorithm and scale with the number of changes instead of the number of nodes under diff. The major improvement presented in our approach is related to the representation and building of the trees that support the diffing. By representing elements of the tree in a structure of arrays (contrary to heap allocated nodes) we were able to improve memory locality, in addition with such contiguous representation we were able to use bit-sets on multiple occasions, instead of usual hash-sets. More importantly, we were able to limit the memory footprint of the top-down and bottom-up phases by skipping decompressions of subtrees from the *HyperAST* (its DAG).

The evaluation showed that our approach outperforms the well-known source code diffing approach presented by *GumTree*. In particular, we observed an order-of-magnitude difference in CPU time from ×1.2 to ×12.7 for the total time of diff computation and up to ×226 in intermediate phases of the diff computation. We also observed an order-of-magnitude difference in memory footprint of ×4.5 per AST node. Finally, we gained all the time while having 99.3% of identical diffs with respect to GumTree and 99.99975% of identical mappings in the remaining 0.7% diffs. When including the parsing cost along with the diff, we still outperformed GumTree-Spoon. We were faster by 14.52 times in half of the cases (median) and when excluding extreme cases of gains, we were faster on average by 13.68 times.

Chapter 9

Implementation and Practical Uses

This chapter presents and discusses implementation and practical uses related to *HyperAST*.

9.1 Implementation Concerns throughout the Thesis

This section will discuss the more technical concern underlying my thesis. Indeed, during my thesis and to support my research, I actually developed two tools and later extended the second. Actually, the second tool was a response to the limitations of the first one and other solutions available at the time. One candidate could have been related to *RefactoringMiner*. However, its orientation towards representing what changes in priority was arguably in conflict with the approach taken in my first publication (chapter 5), where prioritizing what did not change was, in my view, crucial to effectively analyzing code histories.

9.1.1 Limitations in First Approach (chapter 5)

I developed the tooling for my first article (chapter 5) using Java, relying on two state-of-the-art libraries, namely *Spoon* and *Gumtree*. They both work on code trees (in *Spoon* references are resolved on the fly), but I faced unsatisfying trade-offs and limitations. Indeed, in Java there are two ways of structuring data, with primitives or with referential objects (Python and Ocaml suffer from similar issues, and C# to a lesser extent). Other developers have encountered similar limitations, for example, the creator of *JGit*, Shawn O. Pearce, in an email, seems to explain that in the end, one either accepts a large performance hit (for data intensive tasks, easily two times slower) or just abandon most concepts considered good practices related to Object-Oriented Programming (OOP) and ends up using *Java* primitives to write *Java* code that actually resemble C code [1]:

> "JGit struggles with not having unsigned types in Java."
>
> "JGit struggles with not having an efficient way to represent a SHA-1. C can just say "unsigned char[20]" and have it inline into the container's memory allocation. A byte[20] in Java will cost an *additional* 16 bytes of memory, and be slower to access because the bytes themselves are in a different area of memory from the container object. We try to work around it by converting from a byte[20] to 5 ints, but that costs us machine instructions."
>
> "Native Java collection types have been a snare for us in JGit. We've used java.util.* types when they seem to be handy and already solve the data structure problem at hand, but they tend to preform a lot worse than writing a specialized data structure."

[1] https://marc.info/?l=git&m=124111702609723&w=2

There are issues with maintainability regarding performances:

> "Both parts of JGit are about as good as I know how to make them, but we're really at the mercy of the JIT, and changes in the JIT can cause us to perform worse (or better) than before. Unlike in C Git where Linus has done assembler dumps of sections of code and tried to determine better approaches."

Shawn O. Pearce also gives some orders of magnitude:

> "Notably, 'rev-list –objects –all' takes about 2x as long in JGit as it does in C Git on a project like the linux kernel, and 'index-pack' for the full 270M pack file takes about 2x as long."

Finally, even after aggressive optimizations impacting the modularity and maintainability, one might suffer significant performance hits from using *Java*.

> "its practical to build Git in a higher level language, but you just can't get the same performance, or tight memory utilization, that C Git gets."

Actually, there are some attempts at making *Java* better, notably *Valhalla* [2] that aims to provide user defined primitives, or *Apache Arrow* that handles storing columnar data for you. However, *Valhalla* is still far from being ready to be used, and using *Apache Arrow* would require a complete rewrite of AST nodes in *Spoon* and *GumTree*. There is also *GrallVM* that attempts to fix issues related to the *JVM* and *Java*. It marginally improves performances [WW12] and Garbage Collector (GC) but does not solve limitations of Foreign Function Interface (FFI) in Byte-code mode. Moreover, *GrallVM* does not tackle memory layout limitations. Finally, the compilation to native target is exchanging Just In Time (JIT) for acceptable FFI.

9.1.2 Finding Alternatives to *Java*, *Spoon*, and *Gumtree*

During the writing of [Le +21], I found the *GitHub* Semantic project [Git22], that was targeted toward code navigation and name resolution. At this point, it was actively developed using composition of semantics (also called *Data types à la carte*) and a very interesting effect system [3] to process and analyze an AST. However, it is developed in a very advanced Haskell that I was not confident I could comprehend and make it work for my usecase. At some point (during the development of the *HyperAST*), shortly after having shipped a beta on *GitHub*, they actually left aside a large part of the *Data types à la carte* [4] in favor of *stack-graphs* [Gitb] which is their implementation [5] of the formalism of Antwerpen et al. [Ant+18] that specifically focuses on name resolution. I believe they choose to take this route due to major advantages of scope graphs, particularly its incrementally (see section 3.1.2).

While considering to change of programming language (departing from Java), my main requirement was to obtain more control over the memory layout, my secondary requirement was to be able to write more generic and modular code without major trade-offs regarding performances. C++ could have been a good choice regarding these two criteria, yet I was not

[2] https://openjdk.org/projects/valhalla/

[3] https://hackage.haskell.org/package/fused-effects

[4] You can find a branch pointing to the attempt using compositions of semantics https://github.com/github/semantic/tree/algebraic-experiments.

[5] Hendrik van Antwerpen is currently actively developing *stack-graph* at *GitHub*.

confident using templates to make generic code, and properly handling memory. At this point I heard about a system language called Rust, that was leveraging LLVM and had a first edition shipped in 2018, so it seemed robust enough to support a prototype. The deal maker compared to C++ was the more constrained and verified move semantic in the form of the ownership and borrowing concepts. Compared to *Java* and *C++*, switching to *Rust* had multiple advantages: As said earlier, it had better generics (type checks before monomorphisation), better memory control and management, *i.e.,* explicit while safe control over memory layout and compile time garbage collection. *Rust* is also more explicit about type conversion, thus, helping with reducing bugs. It also provides pattern matching akin to functional languages. It provides a first class interoperability with *C*, particularly useful as I was eager to use *Tree-Sitter* [Mic22] which is written in *C* and provides additional features in *Rust*. Finally, *Rust* provides very helpful compilation-error messages, powerful macro (very useful to handle certain genericity limitations), and a modern package manager named *Cargo*.

In the end, I replaced Java with Rust, *JGit* with *git2-rs* (a *Rust* wrapper over *libgit2*), *Spoon* with *HyperAST* and *Tree-Sitter*, *GumTree* with *HyperDiff*.

9.2 Tackling Efficiency Concerns while Implementing *HyperAST* and *HyperDiff*

The *HyperAST* is a DAG stored in an Entity-Component-System database [Nys14], inspired from Silva, Campos, and Rocha [SCR21] that showed high performance gains. **The code of the *HyperAST* implementation is open source and freely available.**[6] The implementation interacts with *Git* to obtain the history of a given repository through its Merkel DAG. The implementation then uses *Tree-Sitter* [Mic22] to parse the code of each *Git Blob*. *Tree-Sitter* is an incremental and resilient parser that tries to fix erroneous CSTs to a certain extent. *Tree-Sitter* is thus able to handle commits with erroneous code that a compiler would not.

The following paragraphs present some "optimizations" used on the *HyperAST*.

Structures of Arrays (SoAs) are widely used in performance sensitive contexts, to reduce memory wasted by padding, and reduce memory contention, *i.e.,* improving spatial and temporal locality. Indeed, it can help massively with reducing cache misses, especially when only a subset of fields is needed. Moreover, to best of my knowledge, there is no readily optimization passes that are able to provide equivalent performance benefits. This layout is used on multiple occasions for the *HyperAST*: For the main store of nodes through an Entity Component System (ECS) [Gil19]. For the decompressed tree used in *HyperDiff*. It could also be used for the DAG of references used of the reference analysis in chapter 7.

A **Bit Set** implements a set using an array of bits. It is limited to dense sets of a limited size, such as with identifiers given by increasing a counter. This structure becomes rapidly more memory efficient than a hash set (a single bit per element) when the number of elements increases. Moreover, considering modern virtual memory, where zeroed pages are not physically allocated,

[6] https://github.com/quentinLeDilavrec/HyperAST

it is also possible to defer physical memory allocation. It might be possible to further improve performances by using *Judy arrays* or *Tries*. I use **Bit Sets** on multiple occasions, especially for *HyperDiff* to implement existential operators used in the *GumTree* algorithm.

Another data structure based on a **Bit Set** that I used with the *HyperAST* is the **Bloom Filter** [Blo70]. I made a simplified implementation for the reference oracle presented in chapter 7. My implementation only uses one hash function shared by the whole *HyperAST*, it would be suboptimal if we considered a single Bloom Filter, but the *HyperAST* is a recursive data structure, such that reduced performances of an oracle are counteracted by the oracles in its children. Exactly quantify this action would require an in depth study, and could very well be applied to features other than reference analysis, typically clone detection.

9.3 Improving Dissemination with a Graphical Demonstrator

To facilitate the diffusion of my research, I developed a graphical demonstrator leveraging the *HyperAST* that enable its users to compute statistics and track pieces of code in a Git history (and publicly available on a Git forge). I was able to demonstrate its use (along with the tracking section 9.3.2) on multiple occasions at the 2023 edition of GDR GPL and at weekly team meetings. For the demonstrations, I often presented the Code base of Spoon (Java), Linux kernel, and Stockfish (C/C++).

The graphical interface is implemented using the *egui* library, it is especially interesting as it enabled me to easily target web browsers and native platforms, it also left me free of implementing complex renderings and interactions that would have been difficult to achieve in the usual web engines. The Representational State Transfer Application Programming Interface (REST API) was implemented using the *axum* library, it is modular, extensible, and leverage the *Rust* type system without abusing macro.

9.3.1 Computing batch metrics

This particular demonstration on computing batch metrics has two objectives: Section 9.3.1 presents how we showed that it was possible to query code stored inside the *HyperAST i.e.,* computing statistic and matching certain patterns (be it very programmatically). Section 9.3.1 presents how we demonstrated the added value of the approach regarding efficiency by comparing the usual approach to one that leverages the *HyperAST*. We limit ourselves to a few commits as the naive method take a long time to execute (we could have made it parallel, but it would only reduce latency). The simple query should be sufficient to show the performance benefits of our tool.

Scripting metrics computation on *HyperAST*

In the Graphical User Interface (GUI) and through a REST API it is possible to dynamically execute arbitrary (sandboxed) queries on the *HyperAST*. It is still very programmatical, *i.e.,*

writing with a Rust syntax and *Javascript* semantic, we leverage the scripting language and interpreter called *Rhai*[7] to traverse the *HyperAST* and compute simple statistics. This scripting system was chosen as it is sandboxed, and configurable/extendable, while still being efficient.

Figure 9.1 presents a very simple query (center) that counts the number of files and number of nodes in ten commits from *Stockfish*[8] starting at 7f2eb10e (left side) going over consecutive commits.

The computing model is transparent to the way it traverses the *HyperAST*, it is written in three parts with data and functions that can be provided in scope:

— **Init** returns the initial state that will be passed to the first **Filter**. Additional accumulator structures are provided to compute statistics (mean, max, quantiles) and format specific data (file path + payload).

— **Filter** selects children than will be traversed and initialize their payload. The payload for the current node is provided in scope as `s`. Functions are also provided, such as `is_directory()`, `is_file()`, and `children()`. The first two are predicates on the type of the current node and `children()` retrieves the list of child ids, so it can return along with their payload.

— **Accumulate** accumulates the payload from current node `s` to its parent `p`. Note that the payload is a dynamic object so it is possible to add specific behaviors and overload operators, such as `+=`.

The right side of fig. 9.1 show metrics returned after running the query on the ten commits. It takes the form of a table with a commit per row, first column is the short id, last is about the compute time (excluding build time of the *HyperAST*), and all other columns contain metrics accumulated in the payload.

Benefits of Using Metadata in *HyperAST*

In fig. 9.1, the total compute time is 5.88 minutes, decomposed into (top right) 1.61s to build the *HyperAST* and 5.86m to compute the metrics on the 10 commits. The last column shows the time taken per commit, between 32 and 40 seconds. Thus, it is made obvious that running the scripts is orders of magnitude slower than building the *HyperAST* *i.e.,* parsing and adding the content of each commit to the *HyperAST*. Then, by running a slight variation of the script, as shown in fig. 9.2, allow us to show the performance benefits of the *HyperAST*. Actually, the script differs in two major places: in the filtering phase it stops at files, and in the accumulation phase, from the current node, using `size()` it accesses the precomputed metadata that stores the number of elements contained in the subtree. The direct consequence of these modifications is that only a fraction of the nodes need to be accessed, around 40 instead of 14 000. We can observe that it takes less than 15 milliseconds per commits (2000 times less than the previous case), yet it still returns the same total number of nodes.

[7] https://github.com/rhaiscript/rhai
[8] https://github.com/official-stockfish/Stockfish

Figure 9.1 – Compute the number of files and number of nodes in 10 commits of *Stockfish*, naively *i.e.*, by traversing the whole Tree for each commit.

Figure 9.2 – Compute the number of files and number of nodes in 10 commits of *Stockfish*, smartly *i.e.*, by using metadata precomputed on subtrees.

9.3.2 Code tracking

The demonstrator also provides a graphical interface to track source code, this time it uses *HyperDiff* [Le +23], notably the mappings, and leverages the top-down phase to take a shortcut when the piece of tracked code is mapped early. Figure 7.7 displays a screenshot of the app working on *Stockfish*–the famous chess engine, where I tracked some configuration variables that set the value of chess pieces. In a few minutes, I was able to track them back to their first introduction, through more than three thousand commits (see "skipped <#commit> commits" on the bottom panes displaying metadata). During the tracking it is possible configure when to stop ("Triggers" in the left menu). In this case, I choose to stop every time the global constant changed. Actually, it also stops when the piece of code failed to be matched, in this case it is possible to recover manually, but it should be possible to further improve the tracking heuristic and reduce the number of manual interventions.

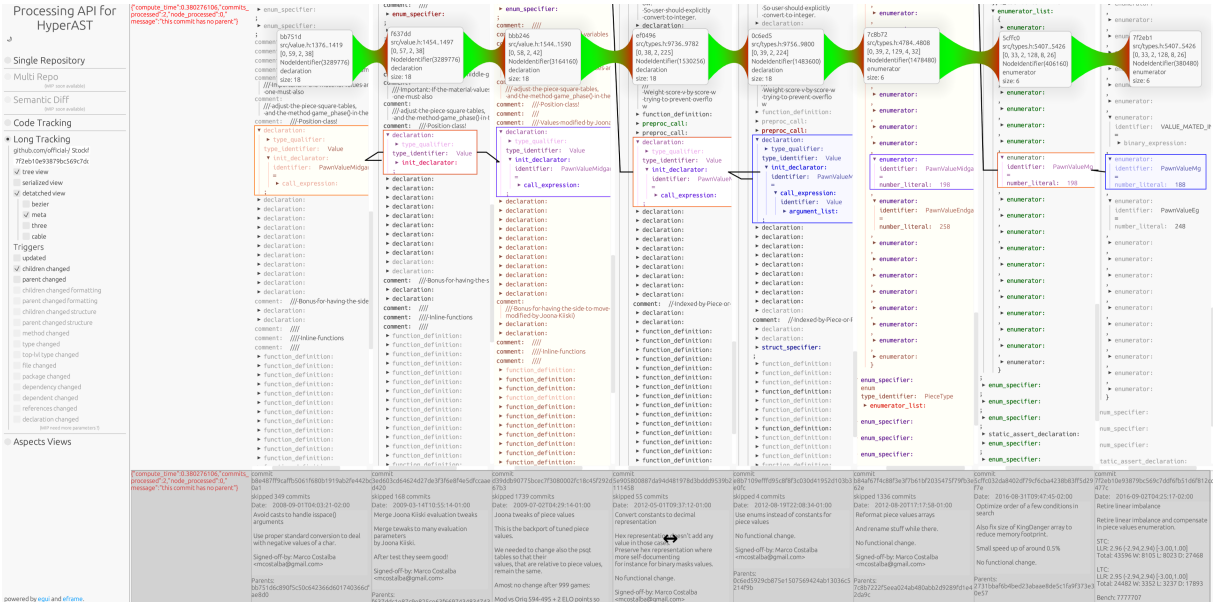


Figure 9.3 – Code tracking.

9.4 Grading programming homework

Together with Djamel Eddine Khelladi, I supervised a group of master student (first year) for a year, each Wednesday afternoons. The given technical goal was to produce a *Visual Studio Code* extension using the *HyperAST* that would help teachers with the grading of programming homework, and it could have been extended to a more industrial context to compare different code histories. The pedagogical goal was to introduce student to modern development assistant tools, such as Integrated Development Environment (IDE) extensions (*e.g., Visual Studio Code* extension), protocols for developer tooling (*e.g.,* LSP), and more fundamentally structured code representation and analysis. The students were able to develop the extension, and demonstrated its principle in front of their fellow students. They showed the process of a professor selecting his repository, then selecting students repositories that should be considered (they were able

to use the *GitLab* API) and selecting the metrics that should be computed. Then after a few seconds to clone, preprocess and compute metrics (with the *HyperAST*), it displayed a table of metrics (columns) by students (rows). They were able to provide some metrics, and count some code patterns. Finally, they also showed an adjacency matrix comparing the distance (Jaccard similarity coefficient over the set of subtrees) between each repository, to help the professor detect signs of gross plagiarism.

Chapter 10

Conclusion and Perspectives

In this thesis, I started by exploring possibilities offered by fine-grained and precise source code analyzes in the form of co-evolutions to understand and maintain source code histories. I especially considered the testing aspect, using test execution verdicts to explain causes for tests evolutions. To make such dynamic analysis scale, I adjoined static analysis technics allowing to preselect groups of evolutions and tests that could lead to a co-evolution thus reducing the combinatorial explosion of cases to dynamically analyze. We evaluated our approach on 45 industrial grade repositories. On this occasion we were able to automatically find 140 complete immediate co-evolutions *i.e.,* breaking then repairing a test, and 500 partial immediate co-evolutions *i.e.,* that did not fully fix the broken tests. More importantly we were able to find 62 delayed co-evolutions spanning over multiple commits *i.e.,* a test is broken only to be repaired in a later commit, and 112 including partials.

Precisely analyzing source code histories with approaches available at the time, allowed me to notice opportunities for orders of magnitude more efficient analyzes, by leveraging spatial and temporal code redundancy along with partial source code analyzes. It led me to devise the *HyperAST*, a framework enabling incremental analysis of source code histories in a fine-grained structured form (AST), by leveraging properties of a DAG. To evaluate the *HyperAST*, we used 18 well known Java projects, and processed thousands of commits. The *HyperAST* was more than 91.8% more memory efficient and 83.4% faster to construct than the usual approach.

I presented concrete benefits of the *HyperAST* for spatial analysis through a novel reference analysis that enables to find all references to a given declaration without having to first compute a global index, nor updating this index after change. It proposes complete change of doing reference analyzes, by partially computing unresolved references to accelerate reference searches. Our reference analysis was evaluated along the *HyperAST*, and showed an average 90% precision and 97% recall compared to the baseline results. We were also able to show that our approach was lazy *i.e.,* a search time proportional to the number of declarations considered, while it showed similar performances to the baseline in the worst case (considering all declaration).

I also presented the benefits of the *HyperAST* for temporal analysis through *HyperDiff*, an incremental structured diff algorithm that leverages unchanged code to reduce memory usage. During the evaluation, we showed that *HyperDiff* computed valid diffs compared to the baseline, on average 4.5 times lower memory footprint, and on average 83% faster than the baseline.

These two analyses introduce two different ways of leveraging the *HyperAST*'s DAG topology. The reference analysis shows a contextual analysis from which non-contextual intermediate computation are extracted, thus making the original analysis incremental. The diff analysis complementarly shows a lazy decompression from the DAG to a tree topology, thus reducing

memory contention while providing a more conventional tree.

I also developed a graphical demonstrator to help with dissemination and facilitate experimentation on the *HyperAST*. It presents a query system that allows to compute statistics, and it also provides code tracking along with alternative code and changes visualization.

10.1 Perspectives and future work

This section presents perspectives to this thesis with possible impacts both on fundamental data structures and practices, and on development tasks automation and quality improvements.

10.1.1 Automated Source Code Co-evolution

The approach presented in chapter 5 already has all the component necessary to mutate source code given a list of actions, they are used to apply sets of evolutions before running test, so it should be straight forward to use it for some degree of automation using very simple heuristics comparable to the one used in mutation testing. Yet, it would still require work to extract and apply complex co-evolution pattern. First, it would necessitate to synthesis co-evolution patterns. Then, matching patterns applicable on a given source code. Finally, it would require to select the best pattern to apply.

Using the *HyperAST*, I believe, would allow a better scaling of the original approach, yet some features would have to be migrated, specifically the ones related to actioning compilation and tests through *Maven*, and the code implementing the co-evolution assembler. It would also require to finish and test a full-fledged impact analysis that uses the *HyperAST*. Indeed, the semantic of references in Java is very complex, so chaining searches can currently produce unsatisfactory results. The main reason I did not invest time perfecting the reference analysis of Java programs is related to recent work of Creager and Antwerpen on stack-graphs [Gitb; CA22] that is the focus of section 10.1.3.

10.1.2 Rewriting source code histories

Source code histories are useful in many circumstances: to collaborate and archive, for code provenance and to track root cause of bugs, to study development processes, to take part in a new project, to train ai systems to fix bugs and write code, or simply by curiosity. Yet, using raw source code histories has pitfalls and might prove itself suboptimal, there is a need for preprocessing. When researchers analyze code histories they often filter certain commits (to evaluate *RefactoringMiner* merges are filtered out, *i.e.,* commits with two parents or more). In general, rewriting a code history that has already been shared is a bad practice, yet sometimes developers have to rewrite specific parts of their code history, notably to remove confidential data or make part of an internal repository public. Moreover, rewriting source code histories could also be used to simplify a code base by removing changes that cannot be traced back to a (recent) release. Rewriting could also be used to reorder commits, to posteriorly adopt (or prepare adopting) good engineering practices like Test Driven Development (TDD), or bring

bug fixes closer (even merging it) to the commit introducing the fixed code. Removing useless changes is something that can be done with approaches, such as commit slicing [Li+17]. Yet it does not model a distance or an order between changes.

I think rewriting source code histories could leverage the approach presented in chapter 5 regarding co-evolutions. Considering the idea of posteriorly adopting (or preparing to adopt) good engineering practices like TDD, we could consider the addition of tests (and their components: assertions, inputs, and calls) as causes of co-evolutions, such that moving the test earlier in time, but just before their repair would make the history look like it was developed in a perfect TDD fashion. Actually rewriting could be very similar to automatically applying co-evolutions if cleverly implemented. It would first require to find co-evolutions, then compute the distance metric between breaks and repairs, correlated to the fragmentation of co-evolutions (*i.e.,* more or less *delayed*), then move the repairs closer to the breaks, or even putting repairs located in tests before the breaks.

10.1.3 Robust reference analysis

A recent work of Creager and Antwerpen on stack-graphs [Gitb; CA22] in addition to [PZH23] have shown that it was possible to declaratively specify the semantic of references of general purpose languages, such as *Java*, *Javascript*, and *Typescript*. While at the same time incrementally resolving them. It opens up new way of implementing robust reference analyzes, and my approach (chapter 7) could benefit from it. Indeed, I had to implement the partial reference analysis in about 10KLoCs and the reference search in about 4KLoCs to show that my approach work. However, extending my implementation would require a lot more work to develop and maintain robust reference analyzes for multiple languages, compared to, I believe, using their declarative specification. To be sure, I would have to make a prototype. Yet, it would first require me to implement the *Tree-Sitter* queries for the *HyperAST*. Then, I would have to find a way to adapt their graph concepts to my partial resolution facilities. I would also have to do adaptations for the search process, which is a total 180 compared to their original intended use (resolution process).

10.1.4 Advanced code tracking

Code tracking is a difficult problem that shares a lot with diffing. For example, *Refactoring-Miner* [TKD20] was used to develop *Code Tracker* [JT22]. Notably, SEAL [Sat+23] leverages LLVM IR to extract data-flow information from source code to better explain commit interactions. Similarly, combining diffs and ref analysis could lead to better code tracking. Indeed, common tree diff approach are essentially working on tree using purely structural comparisons, yet using referential relations it should be possible to infer structurally ambiguous mappings with a high precision *e.g.,* small method without enough entropy to match structurally when renamed. Usually, such approach would be impractical to the prohibitive code of compilation, yet both our approach presented in chapters 7 and 8 would give us opportunities improving accuracy without making it inconveniently slow. Similarly, it should be possible to improve diffing accuracy by

exploring hyperparameters configuring the diff algorithm [MFM23].

10.1.5 Multi-Level MerkleDAGs: Beyond uniform identifier schemes

This is one of the most technical yet fundamental perspective. Uniform identifiers used in Merkel DAGs as intrinsic identifier are very useful at the architectural level, allowing to uniformly store objects in usual key-value stores. Yet, it is not memory efficient for small nodes, such as fine-grained AST elements. Indeed, for a node (*e.g.,* git object) with a very small content the identifier is actually larger than the stored content. So, I would like to work on circumventing this limitation. I especially see two solutions that could be combined:

— Specialize intrinsic identifiers: given certain context-free information, such as the type, height, size, and bytes, use different identifier schemes (*e.g.,* more or less bytes for the hash with a some discriminant). This way it is still an intrinsic identifier, but with a memory footprint more adapted to an optimal topology for the DAG, *i.e.,* reducing the size of identifier used often.

— Decontextualize subtrees: it would be easy to remove labels from leafs and store them in their parents. The labels would simply be reassigned by topological order, when the original subtrees are requested. The idea here is not to remove subtrees (it would make it impossible to recover the subtree corresponding to a given identifier), but to improve cache locality and put the decontextualized subtrees in a cold store.

At this point, I don't know how these approach would behave on large amount of code, nor do I know the hyperparameters that would be best, yet it could open new possibilities of large code storage that would also be efficient to query.

CONTRIBUTIONS SUMMARY

Chronologically, I started my PhD, by making small a state of the art on code co-evolutions, which put me on track of designing an approach to automatically find fine-grained co-evolutions in code histories. This first article [Le +21] was combining change analysis and impact analysis to select candidate co-evolutions later validated by leveraging unit tests. I implemented the associated tool (the two first repositories in the tool list in section 10.3) in *Java* and used multiple existing libraries such as *Spoon*, *Maven*, *GumTree*, and *RefactoringMiner*. During the evaluation, while analyzing large industrial histories with what was available at the time made me realize the limitations of batch analyzes on code histories. At this point, considering that typical source code histories changed mostly in small increment, there should be a way of making change and impact analysis incremental, while drastically reducing the memory consumption. My second article [Le +22] targeted the problem of representing code histories. It showed that it was possible to make an impact analysis significantly more incremental, while requiring a fraction of the memory. The *HyperAST* tool was created on this occasion. My third article [Le +23] showed that computing tree diffs could also be made more incremental, and significantly more efficient, notably for memory layout and accesses. It is implemented as an extension to *HyperAST*, in a module called *HyperDiff*.

10.2 Publications List

This section lists the academic articles that I co-wrote in the context of this thesis.

[Le +21] Quentin Le Dilavrec et al., « Untangling spaghetti of evolutions in software histories to identify code and test co-evolutions », *in*: *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2021, pp. 206–216 (cit. on pp. 38, 40, 55, 74, 87, 121, 132, 133).

[Le +22] Quentin Le Dilavrec et al., « HyperAST: Enabling Efficient Analysis of Software Histories at Scale », *in*: *37th IEEE/ACM International Conference on Automated Software Engineering*, IEEE/ACM, 2022, pp. 1–12 (cit. on pp. 76, 85, 87, 102, 110, 132, 133).

[Le +23] Quentin Le Dilavrec et al., « HyperDiff: Computing Source Code Diffs at Scale », *in*: *31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'23)*, 2023 (cit. on pp. 103, 126, 132, 133).

10.3 Tool List

This section lists the tools that I developed in the context of this thesis.

Co-evolutions [Le +21]

— github.com/quentinLeDilavrec/java-tests-coevolution *(main repository)*

— github.com/quentinLeDilavrec/impact-miner *(incremental impact computing)*

— github.com/quentinLeDilavrec/ICSME2021 *(reproduciton package)*

— github.com/quentinLeDilavrec/astexplorer *(GUI to visualize co-evolutions)*

***HyperAST* + *HyperDiff* [Le +22; Le +23]**

— github.com/HyperAST/HyperAST *(main repository)*

— github.com/quentinLeDilavrec/ASE2022 *(reproduction package)*

— github.com/quentinLeDilavrec/FSE23 *(reproduction package)*

— github.com/quentinLeDilavrec/refsolver *(helper lib. for evaluation)*

— observablehq.com/collection/@quentinledilavrec/evaluation-hyperast *(notebook for evaluation)*

— observablehq.com/@quentinledilavrec/presentation-hyperast-ase22

— observablehq.com/@quentinledilavrec/presentation-hyperdiff-fse23

List of Abbreviations

SCM Source Code Management

VCS Version Control System

SWH SoftWare Heritage

CI Continuous Integration

CD Continuous Deployment

DAG Direct Acyclic Graph

SCCS Source Code Control System

RCS Revision Control System

CVS Concurrent Versioning System

WWCVSND What Would CVS Not Do ?
It comes from a remark by Linux Tovalds about how he designed *Git*.

WWCVSNED What Would CVS Never Ever Do ?
This is the updated version used by L. Torvalds in a presentation at Google in 2007. https://www.youtube.com/watch?v=4XpnKHJAok8

LoC Line of Code

AST Abstract Syntax Tree

CST Concrete Syntax Tree

OOP Object-Oriented Programming

DAG Direct Acyclic Graph

IDE Integrated Development Environment

LSP Language Server Protocol

CRDT Conflict-free Replicated Data Types

FFI Foreign Function Interface

GC Garbage Collector

JIT Just In Time

AOT Ahead Of Time

SoA Structure of Arrays

AoS Array of Structures

ECS Entity Component System

REST API Representational State Transfer Application Programming Interface

TDD Test Driven Development

GUI Graphical User Interface

Chapter 11

Appendix

11.1 Statistics on Large Source Code Histories

This section records some statistics on large source code histories that I mention in the thesis outside the evaluations of contributions. Using `cloc` through `npx` is a simple way of measuring the size of a codebase.

11.1.1 LLVM

```
$ git clone https://github.com/llvm/llvm-project.git
$ cd llvm-project
$ git log -1 5c91b2886f6bf400b60ca7839069839ac3980f8f --format=%ci | cat
2023-11-11 10:57:44 -0500
git rev-list --count 5c91b2886f6bf400b60ca7839069839ac3980f8f
480479
$ npx cloc 5c91b2886f6bf400b60ca7839069839ac3980f8f
Need to install the following packages:
    cloc
Ok to proceed? (y)
    131888 text files.
    120126 unique files.
    15837 files ignored.

1 error:
Line count, exceeded timeout:  /tmp/yGlAcW6iS2/clang/include/clang/Basic/DiagnosticLexKinds.td

github.com/AlDanial/cloc v 1.96  T=160.46 s (748.7 files/s, 188229.6 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C++                          29626         957439        1868516        5538216
LLVM IR                      35727         641211        5932691        3012388
C/C++ Header                 11834         316607         499285        1485146
C                            10522         259818        1602768        1011002
YAML                          4988          68367          64987         888260
Assembly                     10677         477381        1221032         819602
TableGen                      1311          93710          84379         577524
Text                          1334         186074              0         544462
JSON                           111             18              0         420282
Python                        2318          52229          57881         224176
reStructuredText              1970          70767          86744         145715
XML                            130            187            823         143660
CMake                         2166          12895           8495          95363
Objective-C                   1885          20548          32722          72161
Fortran 90                    2028          19428          93531          71315
Markdown                       219          13734            205          56221
HTML                            84           4744            601          44217
Windows Module Definition      203           4757            142          36370
Objective-C++                  546           7130           6686          27116
OpenCL                         588           5369           9243          19723
Starlark                        65           2273          15137          16328
SVG                             27              0             23          13710
Pascal                          55           3181          19674          11568
Perl                            24           2036           1920           7823
CSV                             64              4              0           6935
Bourne Shell                   113           1016           1171           6558
CUDA                           247           2568           8250           6200
OCaml                           38           1441           2395           5051
SWIG                           125           1088             13           4977
Lisp                            23            412            296           3641
```

awk	2	114	100	3525
CSS	22	463	133	2759
TeX	5	191	5	2556
Smalltalk	187	0	3	2298
JavaScript	23	305	953	2286
make	334	788	211	2053
Bourne Again Shell	14	237	377	1807
HLSL	104	541	1932	1753
DOS Batch	60	177	145	1408
Windows Resource File	113	144	45	1181
Expect	10	23	28	715
Fortran 77	66	42	378	649
Scheme	21	41	24	637
TypeScript	11	105	203	618
Julia	1	164	574	597
C#	8	89	107	570
vim script	16	90	119	545
Bazel	13	73	62	447
Lua	6	46	21	342
Jupyter Notebook	3	0	2434	319
Dockerfile	11	94	188	314
MSBuild script	1	0	7	254
Protocol Buffers	5	53	72	233
diff	2	16	197	195
Rust	4	31	21	140
Mathematica	2	23	0	100
MATLAB	11	7	0	55
INI	8	11	0	50
TOML	2	2	0	25
m4	1	7	0	24
Visual Studio Solution	1	1	1	20
Swift	2	6	0	17
D	1	2	0	16
NAnt script	1	0	0	13
Fortran 95	5	15	47	7
Logos	2	4	0	2
SUM:	120126	3230337	11627997	15344240

11.1.2 Linux

```
$ git clone https://github.com/torvalds/linux.git
$ git show a4d7d701121981e3c3fe69ade376fe9f26324161 | grep Date
Date:   Tue Jun 6 06:18:28 2023 -0700
git rev-list --count a4d7d701121981e3c3fe69ade376fe9f26324161
1186465
$ npx cloc a4d7d701121981e3c3fe69ade376fe9f26324161
Need to install the following packages:
  cloc
Ok to proceed? (y)
   80290 text files.
   70888 unique files.
    9435 files ignored.

github.com/AlDanial/cloc v 1.96  T=89.27 s (794.0 files/s, 386327.9 lines/s)
```

Language	files	blank	comment	code
C	32446	3302774	2592126	17034043
C/C++ Header	23566	708780	1361445	7070233
reStructuredText	3339	165090	67625	451118
JSON	549	2	0	376158
YAML	3489	64392	16151	300159
Assembly	1334	48449	101763	233861
Text	1852	28226	0	125417
Bourne Shell	945	29262	19976	114568
make	2811	11069	11937	50859
SVG	74	90	1171	48177
Python	183	8951	7767	44993
Perl	69	7562	5160	37635
Rust	55	1273	8094	7690
yacc	10	710	409	4996
PO File	6	948	1088	3733
lex	10	373	309	2253
C++	12	384	145	2070
Bourne Again Shell	55	392	309	1611
awk	13	238	154	1373

```
CSV                                10             74              0            654
Glade                               1             58              0            603
NAnt script                         2            153              0            537
Cucumber                            1             34             58            198
TeX                                 1              6             74            156
TNSDL                               2             33              0            140
CSS                                 3             41             60            136
Windows Module Definition           2             15              0            113
m4                                  1             15              1             95
Clojure                            33              0              0             75
XSLT                                5             13             26             61
Umka                                1             17              0             44
MATLAB                              1             17             37             35
vim script                          1              3             12             27
Markdown                            1              8              0             25
Ruby                                1              4              0             25
HTML                                1              1              5             10
INI                                 1              1              0              6
sed                                 1              2              5              5
TOML                                1              1              9              2
-------------------------------------------------------------------------------
SUM:                            70888        4379461        4195916       25913894
-------------------------------------------------------------------------------
```

11.1.3 Chromium

I did not clone the entire repository of Chomium, so I cannot use git to count the number of commits. But GitHub displays 1,344,623 in the commit counter on the "main" branch as of 11/11/2023.

```
$ git clone --depth 1 https://github.com/chromium/chromium.git
$ git show --no-patch --format=%ci e09463b55787b38bafe6ffcb307f563ba7fbc3a8 | cat
2023-11-11 17:51:32 +0000
$ npx cloc e09463b55787b38bafe6ffcb307f563ba7fbc3a8
Need to install the following packages:
  cloc
Ok to proceed? (y)
  384678 text files.
  324053 unique files.
  103011 files ignored.

9 errors:
Line count, exceeded timeout:  /tmp/ZIStYpmliR/chrome/browser/file_select_helper_unittest.cc
Line count, exceeded timeout:  /tmp/ZIStYpmliR/chrome/browser/ui/web_applications/web_app_file_handling_browsertest.cc
Line count, exceeded timeout:  /tmp/ZIStYpmliR/chromeos/ash/components/drivefs/drivefs_pinning_manager.cc
Line count, exceeded timeout:  /tmp/ZIStYpmliR/chromeos/ash/components/drivefs/fake_drivefs.cc
Line count, exceeded timeout:  /tmp/ZIStYpmliR/third_party/blink/perf_tests/speedometer21/resources/todomvc/architecture-examples/angularjs/node_modules/angular/
Line count, exceeded timeout:  /tmp/ZIStYpmliR/third_party/chromevox/third_party/sre/sre_browser.js
Line count, exceeded timeout:  /tmp/ZIStYpmliR/third_party/rust/proc_macro2/v1/crate/src/parse.rs
Line count, exceeded timeout:  /tmp/ZIStYpmliR/third_party/webxr_test_pages/webxr-samples/js/webxr-polyfill.js
Line count, exceeded timeout:  /tmp/ZIStYpmliR/third_party/webxr_test_pages/webxr-samples/js/webxr-polyfill.module.js

github.com/AlDanial/cloc v 1.96  T=256.62 s (1262.8 files/s, 191759.1 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C++                          60040        2596400        1714065       13706599
HTML                        100518         468936          96162        4417454
JSON                          6926           1084              0        3685145
C/C++ Header                 51074        1030188        1281508        3478639
XML                          11264         148134          30233        3249582
JavaScript                   19982         397263         742375        2353762
Java                          9223         243144         300142        1321082
Text                         24855          59885              0        1253769
Python                        6038         181708         231264         741077
Objective-C++                 5068         150730         109100         736487
Rust                          2095          47298          87261         645421
TypeScript                    3856          98807         113890         508095
C                             1106          88774         104432         454848
SVG                           3611           9021          12323         293329
Assembly                       213          23750           9796         256464
Markdown                      2750          62714            526         214338
C#                             232          11650          17482         127363
XHTML                         2440           3776           2969         115713
IDL                           2683          16877              1         113892
CSV                             31              6              0          82135
Mojo                          1723          22015          63173          73116
Protocol Buffers              1217          19014          44899          65574
```

Objective-C	175	9849	8931	62192
YAML	1900	4997	1742	59244
CSS	1333	10349	6534	58611
Bazel	489	4831	2943	49179
diff	250	2860	13596	28089
PHP	770	4218	9654	26926
JSON5	79	561	2653	23429
Bourne Shell	472	5280	7270	23350
Windows Module Definition	26	65	81	18975
Starlark	112	1941	2300	17979
Perl	185	3872	4663	17256
reStructuredText	105	6728	3690	16210
CMake	107	1633	1678	15282
SQL	240	676	1219	14836
m4	27	1510	222	12484
DTD	19	2079	2387	9728
make	67	1304	1288	8463
Ruby	34	1078	555	6520
Swift	73	1046	1139	5113
TOML	81	647	764	3719
Kotlin	23	370	839	2934
Pascal	21	1857	9278	2717
Jinja Template	47	175	29	2505
Bourne Again Shell	57	616	662	2455
DenizenScript	56	39	15	2136
XSLT	78	142	86	1930
Vuejs Component	18	202	235	1910
Maven	10	82	46	1701
Groovy	3	170	217	1534
TeX	2	2	11	1486
Dockerfile	23	205	286	1247
Windows Resource File	27	233	551	1059
XSD	9	193	172	967
CoffeeScript	4	118	32	920
WiX source	1	109	78	798
DOS Batch	45	257	249	777
Meson	9	110	73	776
MSBuild script	12	36	33	725
Flatbuffers	16	344	1367	704
JSX	5	112	60	677
Jupyter Notebook	2	0	968	573
Lisp	8	161	229	570
Go	7	70	58	569
yacc	1	49	42	406
Handlebars	10	10	0	403
Elm	2	114	29	399
vim script	6	52	95	191
Visual Studio Solution	7	7	7	177
awk	2	18	12	156
GLSL	5	31	36	137
INI	12	24	1	134
Gradle	1	27	38	131
SCSS	6	24	15	112
Dart	3	20	7	102
ANTLR Grammar	1	33	0	94
Mako	6	29	13	93
PowerShell	2	23	24	87
PlantUML	1	23	0	82
HLSL	3	8	9	63
SWIG	1	14	28	56
TNSDL	1	16	0	48
D	2	10	70	17
sed	2	11	20	17
ProGuard	1	0	6	16
Standard ML	1	1	0	9
Arduino Sketch	1	4	5	8
WebAssembly	2	0	0	8
Properties	1	0	0	5
Gencat NLS	1	0	0	1
SUM:	324053	5752849	5050941	38406091

```
let rows = document.querySelectorAll(".card > div > div > table > tbody > tr");
let data = [...rows.values()].map(x => ({
    orig: x.querySelector("td:nth-child(1)").textContent,
    typ: x.querySelector("td:nth-child(2)").textContent,
    count:
    ↪ parseInt(x.querySelector("td:nth-child(3)").textContent.replaceAll(",",
    ↪ ""))
}));
let acc = acc = data.reduce((acc, x) => {
    if (acc[x.typ] === undefined) {
        acc[x.typ] = x.count
    } else if (x.count > 0) {
        acc[x.typ] += x.count
    };
    return acc
}, Object.create({}));
let counts = Object.entries(acc).map(x => x[1], 0);
let total = counts.reduce((acc, x) => acc + (x > 0 ? x : 0), 0);

console.log("git");console.log(acc.git)
console.log("total");console.log(total)
console.log("git / total");console.log(acc.git / total)
```

Listing 4 – Script gathering origins count on SWH

```
git
195,307,567
total
200,254,083
git / total
0.9752988007740147
```

Listing 5 – Script results gathering origins count on SWH

11.2 SWH Origins

To obtain data on the number of *Git* projects I ran listing 4 statistics from Software Heritage.[1] The output is shown in listing 5, and we observe that 97.5% of open source projects (by origins without deduplication) available on Software heritage are *Git* repositories while most of them are hosted on *GitHub* (188 millions).

[1] https://web.archive.org/web/20230802013447/https://archive.softwareheritage.org/

BIBLIOGRAPHY

[Ach+19] Mathieu Acher et al., *Learning From Thousands of Build Failures of Linux Kernel Configurations*, Technical Report, Inria ; IRISA, June 2019, pp. 1–12, URL: https://inria.hal.science/hal-02147012 (cit. on pp. 5, 13).

[AHH04] Konstantinos Adamopoulos, Mark Harman, and Robert M Hierons, « How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution », *in*: *Genetic and evolutionary computation conference*, Springer, 2004, pp. 1338–1349 (cit. on pp. 45, 46, 49).

[AHS20] Thazin Win Win Aung, Huan Huo, and Yulei Sui, « A literature review of automatic traceability links recovery for software change impact analysis », *in*: *Proceedings of the 28th International Conference on Program Comprehension*, IEEE/ACM, 2020, pp. 14–24 (cit. on p. 74).

[Ale+19] Carol V Alexandru et al., « Redundancy-free analysis of multi-revision software artifacts », *in*: *Empirical Software Engineering* 24.*1* (2019), pp. 332–380 (cit. on pp. 39, 40, 75).

[And+17] Esben Andreasen et al., « A survey of dynamic analysis and test generation for JavaScript », *in*: *ACM Computing Surveys (CSUR)* 50.*5* (2017), p. 66 (cit. on p. 49).

[Ant+18] Hendrik van Antwerpen et al., « Scopes as types », *in*: *Proceedings of the ACM on Programming Languages* 2.*OOPSLA* (2018) (cit. on pp. 30, 32, 121).

[Arn86] Robert S Arnold, « An introduction to software restructuring », *in*: *Tutorial on Software Restructuring* (1986), pp. 1–11 (cit. on p. 28).

[AS13] Miltiadis Allamanis and Charles Sutton, « Mining Source Code Repositories at Massive Scale using Language Modeling », *in*: *The 10th Working Conference on Mining Software Repositories*, IEEE, 2013, pp. 207–216 (cit. on p. 65).

[Asa+13] Muhammad Asaduzzaman et al., « Lhdiff: A language-independent hybrid approach for tracking source code lines », *in*: *2013 IEEE International Conference on Software Maintenance*, IEEE, 2013, pp. 230–239 (cit. on p. 23).

[Atw+21] Hassan Atwi et al., « PyRef: refactoring detection in Python projects », *in*: *2021 IEEE 21st international working conference on source code analysis and manipulation (SCAM)*, IEEE, 2021, pp. 136–141 (cit. on p. 25).

[Bal+97] Thomas Ball et al., « If your version control system could talk », *in*: *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*, vol. 11, 1997 (cit. on p. 38).

[Bav+12] Gabriele Bavota et al., « TraceME: traceability management in eclipse », *in*: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, IEEE, 2012, pp. 642–645 (cit. on p. 74).

[Bel+17] Moritz Beller et al., « Developer testing in the ide: Patterns, beliefs, and behavior », *in*: *IEEE Transactions on Software Engineering* 45.*3* (2017), pp. 261–284 (cit. on p. 34).

[Bla22] Taylor Blau, *Git at GitHub Scale*, https://ttaylorr.com/presentations/git-merge-2022.pdf, Accessed: 2023-09-25, 2022 (cit. on p. 21).

[Blo+18] Arnaud Blouin et al., « User Interface Design Smell: Automatic Detection and Refactoring of Blob Listeners », *in*: *Information and Software Technology* 102 (May 2018), pp. 49–64 (cit. on p. 40).

[Blo70] Burton H Bloom, « Space/time trade-offs in hash coding with allowable errors », *in*: *Communications of the ACM* 13.*7* (1970), pp. 422–426 (cit. on pp. 91, 123).

[BM04] Andrei Broder and Michael Mitzenmacher, « Network applications of bloom filters: A survey », *in*: *Internet mathematics* 1.*4* (2004), pp. 485–509 (cit. on p. 91).

[Bol+20] Paolo Boldi et al., « Ultra-large-scale repository analysis via graph compression », *in*: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2020, pp. 184–194 (cit. on p. 74).

[Bro19] Nicolas Brousse, « The issue of monorepo and polyrepo in large enterprises », *in*: *Companion Proceedings of the 3rd International Conference on the Art, Science, and Engineering of Programming*, 2019, pp. 1–4 (cit. on p. 21).

[CA22] Douglas A Creager and Hendrik van Antwerpen, « Stack graphs: Name resolution at scale », *in*: *arXiv preprint arXiv:2211.01224* (2022) (cit. on pp. 32, 129, 130).

[CAM02] Gregory Cobena, Serge Abiteboul, and Amelie Marian, « Detecting changes in XML documents », *in*: *Proceedings 18th International Conference on Data Engineering*, IEEE, 2002, pp. 41–52 (cit. on pp. 24, 80).

[CCD08] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta, « Tracking your changes: A language-independent approach », *in*: *IEEE software* 26.*1* (2008), pp. 50–57 (cit. on p. 23).

[CDR09] Michel Chilowicz, Etienne Duris, and Gilles Roussel, « Syntax tree fingerprinting for source code similarity detection », *in*: *2009 IEEE 17th international conference on program comprehension*, IEEE, 2009, pp. 243–247 (cit. on p. 80).

[CG97] Sudarshan S Chawathe and Hector Garcia-Molina, « Meaningful change detection in structured data », *in*: *ACM SIGMOD Record* 26.*2* (1997), pp. 26–37 (cit. on p. 24).

[Cha+96] Sudarshan S Chawathe et al., « Change detection in hierarchically structured information », *in*: *Acm Sigmod Record* 25.*2* (1996), pp. 493–504 (cit. on pp. 24, 101, 109, 112, 113).

[Cic+08] Antonio Cicchetti et al., « Automating co-evolution in model-driven engineering », *in*: *Enterprise Distributed Object Computing Conference, 2008. EDOC'08. 12th International IEEE*, IEEE, 2008, pp. 222–231 (cit. on p. 50).

[ČM03] Davor Čubranić and Gail C Murphy, « Hipikat: Recommending pertinent software development artifacts », *in*: *Proceedings of the 25th international Conference on Software Engineering*, IEEE Computer Society, 2003, pp. 408–418 (cit. on pp. 45, 48).

[Cor+16] Benoit Cornu et al., « Casper: Automatic tracking of null dereferences to inception with causality traces », *in*: *Journal of Systems and Software* 122 (2016), pp. 52–62 (cit. on p. 40).

[Dam+17] Anusha Damodaran et al., « A comparison of static, dynamic, and hybrid analysis for malware detection », *in*: *Journal of Computer Virology and Hacking Techniques* 13 (2017), pp. 1–12 (cit. on p. 36).

[Dan+19] Benjamin Danglot et al., « A snowballing literature study on test amplification », *in*: *Journal of Systems and Software* 157 (2019), p. 110398 (cit. on p. 35).

[Dem+16] Andreas Demuth et al., « Co-evolution of metamodels and models through consistent change propagation », *in*: *JSS* 111 (2016), pp. 281–297 (cit. on p. 50).

[DGM10] Brett Daniel, Tihomir Gvero, and Darko Marinov, « On test repair using symbolic execution », *in*: *Proceedings of the 19th international symposium on Software testing and analysis*, ACM, 2010, pp. 207–218 (cit. on pp. 45, 46, 48–50).

[DH08] Barthélémy Dagenais and Laurie Hendren, « Enabling static analysis for partial java programs », *in*: *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, 2008, pp. 313–328 (cit. on p. 32).

[Di +14] Juri Di Rocco et al., « Dealing with the Coupled Evolution of Metamodels and Model-to-text Transformations. », *in*: *ME@ MoDELS*, 2014, pp. 22–31 (cit. on p. 50).

[DIP11] Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio, « What is needed for managing co-evolution in mde? », *in*: *Proceedings of the 2nd International Workshop on Model Comparison in Practice*, ACM, 2011, pp. 30–38 (cit. on p. 50).

[DIP13] Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio, « A methodological approach for the coupled evolution of metamodels and atl transformations », *in*: *ICMT*, Springer, 2013, pp. 60–75 (cit. on p. 50).

[DP16] Georg Dotzler and Michael Philippsen, « Move-optimized source code tree differencing », *in*: *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, IEEE/ACM, 2016, pp. 660–671 (cit. on p. 25).

[DR11] Barthélémy Dagenais and Martin P Robillard, « Recommending adaptive changes for framework evolution », *in*: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20.*4* (2011), p. 19 (cit. on pp. 45, 47–49).

[DR14] Barthélémy Dagenais and Martin P Robillard, « Using traceability links to recommend adaptive changes for documentation evolution », *in*: *IEEE Transactions on Software Engineering* 40.*11* (2014), pp. 1126–1146 (cit. on pp. 45, 47, 49, 76).

[DSK12] Adam Duley, Chris Spandikow, and Miryung Kim, « Vdiff: a program differencing algorithm for Verilog hardware description language », *in*: *Automated Software Engineering* 19.*4* (2012), pp. 459–490 (cit. on p. 24).

[DZ17] Roberto Di Cosmo and Stefano Zacchiroli, « Software heritage: Why and how to preserve software source code », *in*: *iPRES 2017-14th International Conference on Digital Preservation*, 2017, pp. 1–10 (cit. on pp. 5, 6, 13, 14, 21, 23, 85).

[ERS10] Emelie Engström, Per Runeson, and Mats Skoglund, « A systematic review on regression test selection techniques », *in*: *Information and Software Technology* 52.*1* (2010), pp. 14–30 (cit. on p. 36).

[FA14] Gordon Fraser and Andrea Arcuri, « Automated test generation for java generics », *in*: *International Conference on Software Quality*, Springer, 2014, pp. 185–198 (cit. on pp. 46, 49).

[Fal+14] Jean-Rémy Falleri et al., « Fine-grained and accurate source code differencing », *in*: *ACM/IEEE International Conference on Automated Software Engineering, ASE '14*, ACM/IEEE, 2014, pp. 313–324 (cit. on pp. 24, 25, 28, 45, 48, 58, 64, 74, 76, 80, 101–103, 105, 107, 108, 118).

[FGP05] Beat Fluri, Harald C Gall, and Martin Pinzger, « Fine-grained analysis of change couplings », *in*: *Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, IEEE, 2005, pp. 66–74 (cit. on p. 38).

[Flu+07] Beat Fluri et al., « Change distilling: Tree differencing for fine-grained source code change extraction », *in*: *IEEE Transactions on software engineering* 33.*11* (2007), pp. 725–743 (cit. on pp. 24, 40, 45, 48, 118).

[Flu+09] Beat Fluri et al., « Analyzing the co-evolution of comments and source code », *in*: *Software Quality Journal* 17.*4* (2009), pp. 367–394 (cit. on p. 50).

[FMM15] Jean-Rémy Falleri, Matias Martinez, and Martin Monperrus, *Gumtree Spoon*, https://github.com/SpoonLabs/gumtree-spoon-ast-diff, Accessed: 2022-05-15, 2015 (cit. on p. 25).

[Fow99] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, Boston, 1999 (cit. on p. 28).

[FPG03] Michael Fischer, Martin Pinzger, and Harald Gall, « Populating a release history database from version control and bug tracking systems », *in*: *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* IEEE, 2003, pp. 23–32 (cit. on p. 38).

[Fra+18] Nadime Francis et al., « Cypher: An evolving query language for property graphs », *in*: *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1433–1445 (cit. on p. 64).

[Fri+18] Veit Frick et al., « Generating accurate and compact edit scripts using tree differencing », *in*: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2018, pp. 264–274 (cit. on p. 25).

[GA13] Shahabeddin Geravand and Mahmood Ahmadi, « Bloom filter applications in network security: A state-of-the-art survey », *in*: *Computer Networks* 57.*18* (2013), pp. 4047–4064 (cit. on p. 91).

[GAS19] Daniel M German, Bram Adams, and Kate Stewart, « cregit: Token-level blame information in git version control repositories », *in*: *Empirical Software Engineering* 24 (2019), pp. 2725–2763 (cit. on pp. 25, 26).

[GFA13] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri, « Improving search-based test suite generation with dynamic symbolic execution », *in*: *2013 ieee 24th international symposium on software reliability engineering (issre)*, IEEE, 2013, pp. 360–369 (cit. on p. 46).

[GFP09] Harald C Gall, Beat Fluri, and Martin Pinzger, « Change analysis with evolizer and changedistiller », *in*: *IEEE Software 1* (2009), pp. 26–33 (cit. on pp. 45, 48).

[GHJ98] Harald Gall, Karin Hajek, and Mehdi Jazayeri, « Detection of logical coupling based on product release history », *in*: *Proceedings. International Conference on Software Maintenance*, IEEE, 1998, pp. 190–198 (cit. on p. 39).

[Gil19] Thomas Gillen, *Legion*, https://github.com/amethyst/legion, Accessed: 2023-10-20, 2019 (cit. on p. 122).

[Gita] Github, *Github code search*, https://github.blog/2023-02-06-the-technology-behind-githubs-new-code-search/, accessed August 22, 2022 (cit. on p. 31).

[Gitb] Github, *Github Stack-graphs*, https://github.com/github/stack-graphs, accessed July 27, 2022 (cit. on pp. 121, 129, 130).

[Git22] Github, *Semantic*, https://github.com/github/semantic, Accessed: 2022-05-12, 2022 (cit. on pp. 32, 121).

[Gru+21] Felix Grund et al., « CodeShovel: Constructing method-level source code histories », *in*: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, IEEE, 2021, pp. 1510–1522 (cit. on pp. 26, 38, 40).

[Has09] Ahmed E Hassan, « Predicting faults using the complexity of code changes », *in*: *Proceedings of the 31st International Conference on Software Engineering*, IEEE Computer Society, 2009, pp. 78–88 (cit. on pp. 45, 48).

[Hec99] Frank Hecker, « Setting up shop: The business of open-source software », *in*: *IEEE software* 16.*1* (1999), pp. 45–51 (cit. on pp. 6, 14).

[Her+22] Steffen Herbold et al., « A fine-grained data set and analysis of tangling in bug fixing commits », *in*: *Empirical Software Engineering* 27.*6* (2022), p. 125 (cit. on p. 38).

[HH88] William C Hetzel and Bill Hetzel, *The complete guide to software testing*, QED Information Sciences Wellesley, MA, 1988 (cit. on p. 33).

[HHK20] Yoshiki Higo, Shinpei Hayashi, and Shinji Kusumoto, « On tracking Java methods with Git mechanisms », *in*: *Journal of Systems and Software* 165 (2020), p. 110571 (cit. on pp. 25, 26, 38, 40).

[HKB16] Regina Hebig, Djamel Eddine Khelladi, and Reda Bendraou, « Approaches to co-evolution of metamodels and models: A survey », *in*: *IEEE Transactions on Software Engineering* 43.*5* (2016), pp. 396–414 (cit. on pp. 41, 50).

[HM08] Masatomo Hashimoto and Akira Mori, « Diff/TS: A tool for fine-grained structural change analysis », *in*: *2008 15th working conference on reverse engineering*, IEEE, 2008, pp. 279–288 (cit. on p. 24).

[HMK11] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno, « Historage: fine-grained version control system for java », *in*: *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, 2011, pp. 96–100 (cit. on p. 25).

[HO08] William GJ Halfond and Alessandro Orso, « Automated identification of parameter mismatches in web applications », *in*: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, ACM, 2008, pp. 181–191 (cit. on pp. 45, 46, 48, 49).

[HOK17] Yoshiki Higo, Akio Ohtani, and Shinji Kusumoto, « Generating simpler ast edit scripts by considering copy-and-paste », *in*: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2017, pp. 532–542 (cit. on p. 24).

[Hua+18] Kaifeng Huang et al., « Cldiff: generating concise linked code differences », *in*: *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, IEEE/ACM, 2018, pp. 679–690 (cit. on p. 25).

[Jaa+16] Fehmi Jaafar et al., « Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults », *in*: *Empirical Software Engineering* 21.*3* (2016), pp. 896–931 (cit. on p. 67).

[JH10] Yue Jia and Mark Harman, « An analysis and survey of the development of mutation testing », *in*: *IEEE transactions on software engineering* 37.*5* (2010), pp. 649–678 (cit. on p. 35).

[Jia+06] Guofei Jiang et al., « Multiresolution Abnormal Trace Detection Using Varied-Length n-Grams and Automata », *in*: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 37.*1* (2006), pp. 86–97 (cit. on pp. 45, 48, 49).

[JO12] Wei Jin and Alessandro Orso, « BugRedux: reproducing field failures for in-house debugging », *in*: *2012 34th International Conference on Software Engineering (ICSE)*, IEEE, 2012, pp. 474–484 (cit. on pp. 45–47).

[JT22] Mehran Jodavi and Nikolaos Tsantalis, « Accurate method and variable tracking in commit history », *in*: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 183–195 (cit. on pp. 26, 130).

[KB17] Martin Kleppmann and Alastair R Beresford, « A conflict-free replicated JSON datatype », *in*: *IEEE Transactions on Parallel and Distributed Systems* 28.*10* (2017), pp. 2733–2746 (cit. on p. 25).

[KCM07] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic, « A survey and taxonomy of approaches for mining software repositories in the context of software evolution », *in*: *Journal of software maintenance and evolution: Research and practice* 19.*2* (2007), pp. 77–131 (cit. on p. 38).

[Khe+16] Djamel Eddine Khelladi et al., « Metamodel and constraints co-evolution: A semi automatic maintenance of ocl constraints », *in*: *International Conference on Software Reuse*, Springer, 2016, pp. 333–349 (cit. on p. 50).

[Khe+17a] Djamel Eddine Khelladi et al., « A semi-automatic maintenance and co-evolution of OCL constraints with (meta) model evolution », *in*: *Journal of Systems and Software* 134 (2017), pp. 242–260 (cit. on pp. 45–47, 49, 50).

[Khe+17b] Djamel Eddine Khelladi et al., « An Exploratory Experiment on Metamodel-Transformation Co-Evolution », *in*: *Asia-Pacific Software Engineering Conference (APSEC), 2017 24th*, IEEE, 2017, pp. 576–581 (cit. on p. 50).

[Khe+20] Djamel Eddine Khelladi et al., « On the Power of Abstraction: a Model-Driven Co-evolution Approach of Software Code », *in*: *2020 IEEE/ACM 42st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 2020 (cit. on p. 50).

[Kho+12] Foutse Khomh et al., « An exploratory study of the impact of antipatterns on class change-and fault-proneness », *in*: *Empirical Software Engineering* 17.*3* (2012), pp. 243–275 (cit. on p. 39).

[KKE18] Djamel Eddine Khelladi, Roland Kretschmer, and Alexander Egyed, « Change Propagation-based and Composition-based Co-evolution of Transformations with Evolving Metamodels », *in*: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, ACM, 2018, pp. 404–414 (cit. on p. 50).

[KKP07] Sanjeev Khanna, Keshav Kunal, and Benjamin C Pierce, « A formal investigation of diff3 », *in*: *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science: 27th International Conference, New Delhi, India, December 12-14, 2007. Proceedings 27*, Springer, 2007, pp. 485–496 (cit. on p. 25).

[Koy+20] Anil Koyuncu et al., « Fixminer: Mining relevant fix patterns for automated program repair », *in*: *Empirical Software Engineering* 25.*3* (2020), pp. 1980–2024 (cit. on p. 74).

[KSW18] Wael Kessentini, Houari Sahraoui, and Manuel Wimmer, « Automated Co-evolution of Metamodels and Transformation Rules: A Search-Based Approach », *in*: *International Symposium on Search Based Software Engineering*, Springer, 2018, pp. 229–245 (cit. on p. 50).

[Kus+15a] Angelika Kusel et al., « Consistent co-evolution of models and transformations », *in*: *ACM/IEEE 18th MODELS*, 2015, pp. 116–125 (cit. on p. 50).

[Kus+15b] Angelika Kusel et al., « Systematic Co-Evolution of OCL Expressions », *in*: *11th APCCM 2015*, vol. 27, 2015, p. 30 (cit. on p. 50).

[KZ19] Alexander Kampmann and Andreas Zeller, « Bridging the Gap between Unit Test Generation and System Test Generation », *in*: abs/1906.01463 (2019), arXiv: 1906.01463, URL: http://arxiv.org/abs/1906.01463 (cit. on pp. 35, 46, 47, 49, 50).

[Lar+22] Simon Larsén et al., « Spork: Structured Merge for Java with Formatting Preservation », *in*: *IEEE Transactions on Software Engineering* (2022), p. 1, DOI: 10.1109/TSE.2022.3143766 (cit. on p. 25).

[Leh78] Meir M Lehman, *Laws of program evolution-rules and tools for programming management*, 1978 (cit. on pp. 6, 14).

[Leh79] Meir M Lehman, « On understanding laws, evolution, and conservation in the large-program life cycle », *in*: *JSS* 1 (1979), pp. 213–221 (cit. on pp. 6, 14).

[Leh96] Manny M Lehman, « Laws of software evolution revisited », *in*: *Software process technology*, Springer, 1996, pp. 108–124 (cit. on pp. 6, 14).

[LH22] Mateus Lopes and Andre Hora, « How and why we end up with complex methods: a multi-language study », *in*: *Empirical Software Engineering* 27.*5* (2022), pp. 1–42 (cit. on p. 40).

[Li+17] Yi Li et al., « Semantic slicing of software version histories », *in*: *IEEE Transactions on Software Engineering* 44.*2* (2017), pp. 182–201 (cit. on pp. 36, 40, 130).

[Lin+07] Zhiqiang Lin et al., « AutoPaG: towards automated software patch generation with source code root cause identification and repair », *in*: *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, 2007, pp. 329–340 (cit. on p. 40).

[LM18] Julia Lawall and Gilles Muller, « Coccinelle: 10 years of automated evolution in the Linux kernel », *in*: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 601–614 (cit. on pp. 5, 13, 31, 85).

[LR03] Meir M Lehman and Juan F Ramil, « Software evolution—background, theory, practice », *in*: *Information Processing Letters* 88.*1* (2003), pp. 33–44 (cit. on pp. 6, 14).

[LRC22] Yi Li, Julia Rubin, and Marsha Chechik, « Semantic History Slicing », *in*: *Handbook of Re-Engineering Software Intensive Systems into Software Product Lines*, Springer, 2022, pp. 53–77 (cit. on p. 36).

[LS80] Bennett P Lientz and E Burton Swanson, *Software maintenance management*, Addison-Wesley Longman Publishing Co., Inc., 1980 (cit. on p. 26).

[LSC18] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Chen, « A4: Automatically assisting Android API migrations using code examples », *in*: *arXiv preprint arXiv:1812.04894* (2018) (cit. on p. 50).

[LY17] Stanislav Levin and Amiram Yehudai, « The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes », *in*: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2017, pp. 35–46 (cit. on pp. 40, 54, 55, 74, 87).

[LZP09] Zeeger Lubsen, Andy Zaidman, and Martin Pinzger, « Using association rules to study the co-evolution of production & test code », *in*: *2009 6th IEEE International Working Conference on Mining Software Repositories*, IEEE, 2009, pp. 151–154 (cit. on p. 55).

[Maz+16] Davood Mazinanian et al., « JDeodorant: clone refactoring », *in*: *Proceedings of the 38th international conference on software engineering companion*, IEEE/ACM, 2016, pp. 613–616 (cit. on p. 74).

[Mem08] Atif M Memon, « Automatically repairing event sequence-based GUI test suites for regression testing », *in*: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 18.*2* (2008), p. 4 (cit. on pp. 45–50).

[Men02] Tom Mens, « A state-of-the-art survey on software merging », *in*: *IEEE transactions on software engineering* 28.*5* (2002), pp. 449–462 (cit. on p. 25).

[Men08] Tom Mens, *Introduction and roadmap: History and challenges of software evolution*, Springer, 2008 (cit. on p. 74).

[Mer87] Ralph C Merkle, « A digital signature based on a conventional encryption function », *in*: *Conference on the theory and application of cryptographic techniques*, Springer, 1987, pp. 369–378 (cit. on p. 21).

[MES02] David MacKenzie, Paul Eggert, and Richard Stallman, « Comparing and Merging Files with GNU diff and patch », *in*: *Network Theory Ltd* 4 (2002) (cit. on pp. 23, 25).

[Met23] Meta, *Sapling*, https://engineering.fb.com/2022/11/15/open-source/sapling-source-control-scalable/, Accessed: 2023-09-20, 2023 (cit. on pp. 6, 14, 21).

[MFM23] Matias Martinez, Jean-Rémy Falleri, and Martin Monperrus, « Hyperparameter optimization for ast differencing », *in*: *IEEE Transactions on Software Engineering* 49.*10* (2023), pp. 4814–4828 (cit. on p. 131).

[MHK19] Junnosuke Matsumoto, Yoshiki Higo, and Shinji Kusumoto, « Beyond gumtree: a hybrid approach to generate edit scripts », *in*: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, IEEE, 2019, pp. 550–554 (cit. on p. 24).

[Mic] Microsoft, *Language Server Protocol*, Microsoft (cit. on p. 32).

[Mic22] Microsoft, *Tree-Sitter*, https://tree-sitter.github.io/tree-sitter/, Accessed: 2022-05-06, 2022 (cit. on pp. 32, 76, 122).

[MM19] Matias Martinez and Martin Monperrus, « Coming: a tool for mining change pattern instances from git commits », *in*: *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, IEEE Press, 2019, pp. 79–82 (cit. on p. 48).

[MMP13] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman, « Efficient JavaScript mutation testing », *in*: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, IEEE, 2013, pp. 74–83 (cit. on p. 49).

[MMP15] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman, « JSeft: Automated JavaScript unit test generation », *in*: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, 2015, pp. 1–10 (cit. on pp. 46, 49).

[MMP16] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman, « Atrina: Inferring unit oracles from GUI test cases », *in*: *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, 2016, pp. 330–340 (cit. on pp. 46, 49).

[Moh+09] Naouel Moha et al., « Decor: A method for the specification and detection of code and design smells », *in*: *IEEE Transactions on Software Engineering* 36.*1* (2009), pp. 20–36 (cit. on p. 39).

[MP11] Pall Melsted and Jonathan K Pritchard, « Efficient counting of k-mers in DNA sequences using a bloom filter », *in*: *BMC bioinformatics* 12.*1* (2011), pp. 1–7 (cit. on p. 91).

[MPP14] Mehdi Mirzaaghaei, Fabrizio Pastore, and Mauro Pezzè, « Automatic test case evolution », *in*: *Software Testing, Verification and Reliability* 24.*5* (2014), pp. 386–411 (cit. on pp. 45, 46, 48–50).

[MRZ14] Cosmin Marsavina, Daniele Romano, and Andy Zaidman, « Studying fine-grained co-evolution patterns of production and test code », *in*: *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, IEEE, 2014, pp. 195–204 (cit. on p. 50).

[MT04] Tom Mens and Tom Tourwé, « A survey of software refactoring », *in*: *IEEE Transactions on software engineering* 30.*2* (2004), pp. 126–139 (cit. on p. 28).

[Mye86] Eugene W Myers, « An O(ND) difference algorithm and its variations », *in*: *Algorithmica* 1.*1* (1986), pp. 251–266 (cit. on p. 23).

[Ngu+11] Hoan Anh Nguyen et al., « Clone management for evolving software », *in*: *IEEE transactions on software engineering* 38.*5* (2011), pp. 1008–1026 (cit. on p. 24).

[NHM20] Yusuf Sulistyo Nugroho, Hideaki Hata, and Kenichi Matsumoto, « How different are different diff algorithms in git? », *in*: *Empirical Software Engineering* 25.*1* (2020), pp. 790–823 (cit. on p. 23).

[Ni+20] Zhen Ni et al., « Analyzing bug fix for automatic bug cause classification », *in*: *Journal of Systems and Software* 163 (2020), p. 110538 (cit. on p. 40).

[Nys14] Robert Nystrom, *Game programming patterns*, Genever Benning, 2014 (cit. on p. 122).

[Opd92] William F Opdyke, « Refactoring object-oriented frameworks », *in*: (1992) (cit. on pp. 27, 28).

[PA11] Mateusz Pawlik and Nikolaus Augsten, « RTED: A robust algorithm for the tree edit distance », *in*: *Proceedings of the VLDB Endowment* 5.*4* (2011), pp. 334–345 (cit. on pp. 24, 102).

[Pal+14] Fabio Palomba et al., « Mining version histories for detecting code smells », *in*: *IEEE Transactions on Software Engineering* 41.*5* (2014), pp. 462–489 (cit. on p. 40).

[Pal+17] Fabio Palomba et al., « Toward a smell-aware bug prediction model », *in*: *IEEE Transactions on Software Engineering* 45.*2* (2017), pp. 194–218 (cit. on pp. 31, 74, 87).

[Pal+18] Fabio Palomba et al., « On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation », *in*: *Empirical Software Engineering* 23.*3* (2018), pp. 1188–1221 (cit. on p. 39).

[Pap+19] Mike Papadakis et al., « Mutation testing advances: an analysis and survey », *in*: *Advances in Computers*, vol. 112, Elsevier, 2019, pp. 275–378 (cit. on p. 35).

[Par94] David Lorge Parnas, « Software aging », *in*: *Proceedings of 16th International Conference on Software Engineering*, IEEE, 1994, pp. 279–287 (cit. on pp. 6, 14).

[Paw+15] Renaud Pawlak et al., « Spoon: A Library for Implementing Analyses and Transformations of Java Source Code », *in*: *Software: Practice and Experience* 46 (2015), pp. 1155–1179 (cit. on pp. 25, 39, 64, 75, 81).

[PL16] Rachel Potvin and Josh Levenberg, « Why Google stores billions of lines of code in a single repository », *in*: *Communications of the ACM* 59.*7* (2016), pp. 78–87 (cit. on pp. 6, 14, 21, 23).

[PLM06] Yoann Padioleau, Julia L Lawall, and Gilles Muller, « Understanding collateral evolution in Linux device drivers », *in*: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, 2006, pp. 59–71 (cit. on pp. 40, 50).

[Pre+07] William H Press et al., *Numerical recipes 3rd edition: The art of scientific computing*, Cambridge university press, 2007 (cit. on p. 62).

[Pre13] Tom Preston-Werner, « Semantic versioning 2.0. 0 (June 2013) », *in*: *URL https://semver. org* (2013) (cit. on p. 27).

[PZH23] Casper Bach Poulsen, Aron Zwaan, and Paul Hübner, « A Monadic Framework for Name Resolution in Multi-phased Type Checkers », *in*: (2023) (cit. on pp. 32, 87, 130).

[Rap+04] D Rapu et al., « Using history information to improve design flaws detection », *in*: *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.* IEEE, 2004, pp. 223–232 (cit. on pp. 38–40, 74).

[Rei08] Steven P Reiss, « Tracking source locations », *in*: *Proceedings of the 30th international conference on Software engineering*, IEEE, 2008, pp. 11–20 (cit. on p. 23).

[Rob+11] Brian Robinson et al., « Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs », *in*: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, IEEE, 2011, pp. 23–32 (cit. on pp. 46, 50).

[Roc75] Marc J Rochkind, « The source code control system », *in*: *IEEE transactions on Software Engineering 4* (1975), pp. 364–370 (cit. on pp. 6, 14, 19).

[RVV14] Steven Raemaekers, Arie Van Deursen, and Joost Visser, « Semantic versioning versus breaking changes: A study of the maven repository », *in*: *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, IEEE, 2014, pp. 215–224 (cit. on p. 27).

[SA93] Zhong Shao and Andrew W Appel, « Smartest recompilation », *in*: *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1993, pp. 439–450 (cit. on p. 36).

[SAB18] Davide Spadini, Maurício Aniche, and Alberto Bacchelli, « Pydriller: Python framework for mining software repositories », *in*: *Proceedings of the 2018 26th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 908–911 (cit. on p. 39).

[Sat+23] Florian Sattler et al., « SEAL: Integrating Program Analysis and Repository Mining », *in*: *ACM Transactions on Software Engineering and Methodology* (2023) (cit. on pp. 26, 130).

[Sch+15] Johannes Schönböck et al., « Model-Driven Co-evolution for Agile Development », *in*: *System Sciences (HICSS), 48th Hawaii International Conference on*, IEEE, 2015, pp. 5094–5103 (cit. on p. 50).

[SCR21] Pablo Silva, Rodrigo Oliveira Campos, and Carla Rocha, « OSS Scripting System for Game Development in Rust », *in*: *17th IFIP International Conference on Open Source Systems (OSS)*, Springer International Publishing, 2021, pp. 51–58 (cit. on p. 122).

[Sha+11] Marc Shapiro et al., « Conflict-free replicated data types », *in*: *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*, Springer, 2011, pp. 386–400 (cit. on p. 25).

[Sil+15] Luciana L Silva et al., « Developers' perception of co-change patterns: An empirical study », *in*: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2015, pp. 21–30 (cit. on p. 50).

[Sil+20] Danilo Silva et al., « RefDiff 2.0: A Multi-language Refactoring Detection Tool », *in*: *IEEE Transactions on Software Engineering* (2020) (cit. on pp. 25, 40, 45, 48).

[Soe+16] Quinten David Soetens et al., « Change-based test selection: an empirical evaluation », *in*: *Empirical software engineering* 21 (2016), pp. 1990–2032 (cit. on pp. 32, 36).

[Sun+01] Gerson Sunyé et al., « Refactoring UML models », *in*: *UML2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, Springer, 2001, pp. 134–148 (cit. on p. 28).

[SVM19] Luciana L Silva, Marco Tulio Valente, and Marcelo A Maia, « Co-change patterns: A large scale empirical study », *in*: *Journal of Systems and Software* 152 (2019), pp. 196–214 (cit. on p. 50).

[Swa76] E Burton Swanson, « The dimensions of maintenance », *in*: *Proceedings of the 2nd international conference on Software engineering*, IEEE Computer Society Press, 1976, pp. 492–497 (cit. on pp. 6, 14, 26).

[ŚZZ05] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller, « When do changes induce fixes? », *in*: *Proceedings of the 2005 international workshop on Mining software repositories*, 2005, pp. 1–5 (cit. on p. 38).

[Thu+09] Suresh Thummalapenta et al., « MSeqGen: Object-oriented unit-test generation via mining source code », *in*: *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ACM, 2009, pp. 193–202 (cit. on p. 46).

[Tic62] Walter F Tichy, « Design, Implementation and Evaluation of a Revision Control System », *in*: (1962) (cit. on p. 20).

[Tic85] Walter F Tichy, « RCS—A system for version control », *in*: *Software: Practice and Experience* 15.7 (1985), pp. 637–654 (cit. on p. 20).

[TKD20] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig, « RefactoringMiner 2.0 », *in*: *IEEE Transactions on Software Engineering* 1.1 (2020), p. 1 (cit. on pp. 25, 28, 40, 58, 64, 74, 87, 130).

[Tsa+18a] Nikolaos Tsantalis et al., « Accurate and efficient refactoring detection in commit history », *in*: *Proceedings of the 40th International Conference on Software Engineering*, ACM, 2018, pp. 483–494 (cit. on pp. 28, 58, 64).

[Tsa+18b] Nikolaos Tsantalis et al., « Accurate and efficient refactoring detection in commit history », *in*: *Proceedings of the 40th International Conference on Software Engineering*, ACM, 2018, pp. 483–494 (cit. on pp. 45, 48).

[TTN14] Paolo Tonella, Roberto Tiella, and Cu Duy Nguyen, « Interpolated n-grams for model based testing », *in*: *Proceedings of the 36th International Conference on Software Engineering*, ACM, 2014, pp. 562–572 (cit. on pp. 46, 49).

[Tuf+15] Michele Tufano et al., « When and why your code starts to smell bad », *in*: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, IEEE, 2015, pp. 403–414 (cit. on p. 74).

[Tuf+17] Michele Tufano et al., « When and why your code starts to smell bad (and whether the smells go away) », *in*: *IEEE Transactions on Software Engineering* 43.*11* (2017), pp. 1063–1088 (cit. on pp. 39, 74, 87).

[Vau20] Adam Vaughan, *Code kept on ice for 1000 years*, 2020 (cit. on pp. 5, 13).

[VP18] László Vidács and Martin Pinzger, « Co-evolution analysis of production and test code by learning association rules of changes », *in*: *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, IEEE, 2018, pp. 31–36 (cit. on p. 50).

[WLL21] Xiuheng Wu, Mengyang Li, and Yi Li, « EVOME: A software evolution management engine based on differential factbase », *in*: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2021, pp. 1252–1256 (cit. on p. 40).

[Woh+12] Claes Wohlin et al., *Experimentation in software engineering*, Springer Science & Business Media, 2012 (cit. on p. 117).

[WW12] Christian Wimmer and Thomas Würthinger, « Truffle: a self-optimizing runtime system », *in*: *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, ACM, 2012, pp. 13–14 (cit. on p. 121).

[WZL21] Xiuheng Wu, Chenguang Zhu, and Yi Li, « Diffbase: A differential factbase for effective software evolution management », *in*: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 503–515 (cit. on pp. 33, 40).

[YM13] Liguo Yu and Alok Mishra, « An empirical study of Lehman's law on software quality evolution », *in*: (2013) (cit. on pp. 6, 14).

[Zai+08] Andy Zaidman et al., « Mining software repositories to study co-evolution of production & test code », *in*: *2008 1st international conference on software testing, verification, and validation*, IEEE, 2008, pp. 220–229 (cit. on pp. 34, 48, 50, 54, 55, 76).

[Zai+11] Andy Zaidman et al., « Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining », *in*: *Empirical Software Engineering* 16.*3* (2011), pp. 325–364 (cit. on pp. 38, 48, 50, 55, 65).

[Zar20] JetBrains Research Zarina Kurbatova, *kotlinRMiner*, https://github.com/JetBrains-Research/kotlinRMiner, Accessed: 2023-10-20, 2020 (cit. on p. 25).

[ZED11] Pingyu Zhang, Sebastian Elbaum, and Matthew B Dwyer, « Automatic generation of load tests », *in*: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, IEEE Computer Society, 2011, pp. 43–52 (cit. on pp. 47, 49).

[ZH02] Andreas Zeller and Ralf Hildebrandt, « Simplifying and isolating failure-inducing input », *in*: *IEEE Transactions on Software Engineering* 28.*2* (2002), pp. 183–200 (cit. on p. 36).

[ZHM97] Hong Zhu, Patrick AV Hall, and John HR May, « Software unit test coverage and adequacy », *in*: *Acm computing surveys (csur)* 29.*4* (1997), pp. 366–427 (cit. on p. 35).

[Zim+05] Thomas Zimmermann et al., « Mining version histories to guide software changes », *in*: *IEEE Transactions on software engineering* 31.*6* (2005), pp. 429–445 (cit. on p. 39).

[ZS89] Kaizhong Zhang and Dennis Shasha, « Simple fast algorithms for the editing distance between trees and related problems », *in*: *SIAM journal on computing* 18.*6* (1989), pp. 1245–1262 (cit. on pp. 23, 24, 102, 105).

COLLEGE DOCTORAL BRETAGNE
MATHS, TELECOMS
INFORMATIQUE, SIGNAL
SYSTEMES, ELECTRONIQUE

Université de Rennes

Titre : Analyse précise et à l'échelle des historiques de code source

Mot clés : génie logiciel, analyse d'historiques, coévolution, maintenance des logiciels, calcul intensif

Résumé : Les systèmes de contrôle de versions sont l'une des pierres angulaires du développement des logiciels modernes. En intégrant les modifications entre les versions, ils facilitent la collaboration entre les développeurs et permettent l'intégration et la livraison continues des artefacts logiciels. Au fil du temps, les développeurs produisent de nouvelles versions qui introduisent des fonctionnalités et du code supplémentaires. Par exemple, le noyau Linux compte aujourd'hui plus d'un million de commits, 20 000 fonctionnalités et plus de 15 millions de lignes de code. Ce qui ralentit le téléchargement, la construction et l'analyse des logiciels. Par conséquent, il devient essentiel d'améliorer l'efficacité des systèmes qui gèrent les historiques des codes sources des logiciels tout en améliorant la facilité d'utilisation et notre compréhension des historiques déjà existants.

Fournir en permanence des produits et des services robustes sur des systèmes logiciels massifs et complexes nécessite des analyses et des tests approfondis. Par conséquent, l'exécution de toutes les analyses et de tous les tests à chaque changement constitue un défi, voire est carrément irréalisable. Les systèmes de construction et les outils d'analyse existants s'attaquent déjà à cette complexité par des approches incrémentales, soit en modifiant de manière incrémentale un état transitoire, soit en mettant en cache des artefacts intermédiaires. Cependant, les approches existantes considèrent toujours des artefacts à gros grain (souvent au niveau du fichier) et une représentation basée sur le texte (peu d'accès à la sémantique). Dans cette thèse, nous démontrons qu'il est possible d'améliorer de manière significative les approches incrémentales en considérant des éléments à grain plus fin tout en adaptant les analyses existantes.

Cette thèse vise à relever plusieurs défis scientifiques concernant l'extensibilité des approches et des analyses logicielles à de grands projets logiciels industriels. Le premier défi que j'aborde concerne l'analyse temporelle de l'historique des codes sources par le biais de coévolutions fines. Le deuxième défi concerne l'inefficacité des analyses temporelles actuelles du code source, qui ne parviennent pas à exploiter la redondance temporelle et spatiale du code source dans l'historique des logiciels. En effet, dans la plupart des historiques de logiciels, la quantité de changements apportés avant chaque validation est faible par rapport à la taille de l'ensemble de la base de code, c'est-à-dire quelques lignes de code parmi des millions.

Cette thèse présente notament une nouvelle structure de données qui permet de partager le code inchangés finement et de persiter des resultats intermediaires pour ainsi rendre incrementales des analyses temporelles de code source. Ensuite, sont présenté deux analyses, l'une spatial, permettant depuis une déclration de retrouver ces references, l'autre temporel, permettant d'inferer les changements entre deux versions.

Title: Precise temporal analysis of source code histories at scale

Keywords: software engineering, history analysis, co-evolution, software maintenance, performances

Abstract:

VCS are one of the corner stones of modern software development. By tacking changes across versions, they facilitate developers collaboration, and enable continuous integration and delivery of software artifacts. Over time, developers produce new versions that introduce additional features and code. For example, the Linux kernel now has more than 1M commits, 20K features, and more than 15M lines of code. Making it slow to download, build and analyze. Therefore, it becomes essential to improve the efficiency of systems that manage software source code histories while also improving usability and our understanding of already existing histories.

Continuously delivering robust products and services on massive and complex software systems requires extensive analyses and tests. Therefore, executing every analyses and tests at every change is challenging or downright impractical. Existing build systems and analysis tools already tackle this complexity through incremental approaches, either incrementally mutating a transient state or caching intermediate artifacts. Yet, existing approaches still consider coarse grained artifacts (often at file level) and text based representation (little access to semantic). In this thesis, we calm that it is possible to significantly improve incremental approaches by considering finer grained elements while adapting existing analyses.

This thesis aims to tackle various scientific challenges around the scalability of software approaches and analyzes to large industrial software projects. The initial challenge I address concerns the temporal analysis of source code histories through fine-grained co-evolutions, to help with code understanding and automation. The second challenge concerns the inefficiency of current temporal source code analyzes, which fail to exploit temporal and spatial source code redundancy in software histories. In most software histories, the quantity of changes each commit brings is small compared to the size of the whole code base, *e.g.,* a single line in millions of lines of code.

In particular, this thesis presents a new data structure that allows fine-grained sharing of unchanged code and persistence of intermediate results, thus making temporal analyses of source code incremental. Next, two analyses are presented, one spatial, making it possible to retrieve references from a declination, and the other temporal, making it possible to infer changes between two versions.