Technical and Methodological Improvements to Mining Software Repositories

Johannes Härtel

Genehmigte Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.),
Fachbereich 4: Informatik, Universität Koblenz

Vorsitzende des Promotionsausschusses:	Prof. Dr. Karin Harbusch
Berichterstatter:	Prof. Dr. Ralf Lämmel
	Prof. Dr. Alexander Serebrenik
	Prof. Dr. Stefano Zacchiroli
Datum der wissenschaftlichen Aussprache:	September 4, 2023

January 31, 2024

Abstract

Empirical studies in software engineering use software repositories as data sources to understand software development. Repository data is either used to answer questions that guide the decision-making in the software development, or to provide tools that help with practical aspects of developers' everyday work. Studies are classified into the field of Empirical Software Engineering (ESE), and more specifically into Mining Software Repositories (MSR).

Studies working with repository data often focus on their results. Results are statements or tools, derived from the data, that help with practical aspects of software development. This thesis focuses on the methods and high-order methods used to produce such results. In particular, we focus on incremental methods to scale the processing of repositories, declarative methods to compose a heterogeneous analysis, and high-order methods used to reason about threats to methods operating on repositories. We summarize this as technical and methodological improvements. We contribute the improvements to methods and high-order methods in the context of MSR/ESE to produce future empirical results more effectively. We contribute the following improvements.

We propose a method to improve the scalability of functions that abstract over repositories with high revision count in a theoretically founded way. We use insights on abstract algebra and program incrementalization to define a core interface of high-order functions that compute scalable static abstractions of a repository with many revisions. We evaluate the scalability of our method by benchmarks, comparing a prototype with available competitors in MSR/ESE.

We propose a method to improve the definition of functions that abstract over a repository with a heterogeneous technology stack, by using concepts from declarative logic programming and combining them with ideas on megamodeling and linguistic architecture. We reproduce existing ideas on declarative logic programming with languages close to Datalog, coming from architecture recovery, source code querying, and static program analysis, and transfer them from the analysis of a homogeneous to a heterogeneous technology stack. We provide a prove-of-concept of such method in a case study.

We propose a high-order method to improve the disambiguation of threats to methods used in MSR/ESE. We focus on a better disambiguation of threats, operationalizing reasoning about them, and making the implications to a valid data analysis methodology explicit, by using simulations. We encourage researchers to accomplish their work by implementing 'fake' simulations of their MSR/ESE scenarios, to operationalize relevant insights about alternative plausible results, negative results, potential threats and the used data analysis methodologies. We prove that such way of simulation-based testing contributes to the disambiguation of threats in published MSR/ESE research.

Zusammenfassung

Empirische Studien in der Softwaretechnik verwenden Software-Repositories als Datenquellen, um die Softwareentwicklung zu verstehen. Repository-Daten werden entweder verwendet, um Fragen zu beantworten, die die Entscheidungsfindung in der Softwareentwicklung leiten, oder um Werkzeuge bereitzustellen, die bei praktischen Aspekten der Entwicklung helfen. Studien werden in die Bereiche Empirical Software Engineering (ESE) und Mining Software Repositories (MSR) eingeordnet.

Häufig konzentrieren sich Studien, die mit Repository-Daten arbeiten, auf deren Ergebnisse. Ergebnisse sind aus den Daten abgeleitete Aussagen oder Werkzeuge, die bei der Softwareentwicklung helfen. Diese Dissertation konzentriert sich hingegen auf die Methoden und High-Order-Methoden, die verwendet werden, um solche Ergebnisse zu erzielen. Insbesondere konzentrieren wir uns auf inkrementelle Methoden, um die Verarbeitung von Repositories zu skalieren, auf deklarative Methoden, um eine heterogene Analyse durchzuführen, und auf High-Order-Methoden, die verwendet werden, um Bedrohungen für Methoden, die auf Repositories arbeiten, zu operationalisieren. Wir fassen dies als technische und methodische Verbesserungen zusammen um zukünftige empirische Ergebnisse effektiver zu produzieren. Wir tragen die folgenden Verbesserungen bei.

Wir schlagen eine Methode vor, um die Skalierbarkeit von Funktionen, welche über Repositories mit hoher Revisionszahl abstrahieren, auf theoretisch fundierte Weise zu verbessern. Wir nutzen Erkenntnisse aus abstrakter Algebra und Programminkrementalisierung, um eine Kernschnittstelle von Funktionen höherer Ordnung zu definieren, die skalierbare statische Abstraktionen eines Repositorys mit vielen Revisionen berechnen. Wir bewerten die Skalierbarkeit unserer Methode durch Benchmarks, indem wir einen Prototyp mit MSR/ESE Wettbewerbern vergleichen.

Wir schlagen eine Methode vor, um die Definition von Funktionen zu verbessern, die über ein Repository mit einem heterogenen Technologie-Stack abstrahieren, indem Konzepte aus der deklarativen Logikprogrammierung verwendet werden, und mit Ideen zur Megamodellierung und linguistischen Architektur kombiniert werden. Wir reproduzieren bestehende Ideen zur deklarativen Logikprogrammierung mit Datalog-nahen Sprachen, die aus der Architekturwiederherstellung, der Quellcodeabfrage und der statischen Programmanalyse stammen, und übertragen diese aus der Analyse eines homogenen auf einen heterogenen Technologie-Stack. Wir liefern einen Proof-of-Concept einer solchen Methode in einer Fallstudie.

Wir schlagen eine High-Order-Methode vor, um die Disambiguierung von Bedrohungen für MSR/ESE Methoden zu verbessern. Wir konzentrieren uns auf eine bessere Disambiguierung von Bedrohungen durch Simulationen, indem wir die Argumentation über Bedrohungen operationalisieren und die Auswirkungen auf eine gültige Datenanalysemethodik explizit machen. Wir ermutigen Forschende, „gefälschte“ Simulationen ihrer MSR/ESE-Szenarien zu erstellen, um relevante Erkenntnisse über alternative plausible Ergebnisse, negative Ergebnisse, potenzielle Bedrohungen und die verwendeten Datenanalysemethoden zu operationalisieren. Wir beweisen, dass eine solche Art des simulationsbasierten Testens zur Disambiguierung von Bedrohungen in der veröffentlichten MSR/ESE-Forschung beiträgt.

Acknowledgements

I thank my family, especially, Alexander, Christine, Linda and Lukas, for having patience with me, and for helping in many other areas of my life. I thank Ralf for giving me the chance to work freely, without distractions, and without imposing any limitations on thinking. I thank Lukas, Marcel, Daniel and Philipp for sharing the same enthusiasm for science, and for all the discussions. I thank the Koblenz friends, Olaf, AK, Lubosz, Henny, Stef, Alex (and many others) for the past and future parties, and for the company.

Short Biography

- 2008 - 2013 B.Sc. Computer Science at University of Koblenz-Landau.
- 2013 - 2016 M.Sc. Computer Science at University of Koblenz-Landau.
- 2016 - 2022 Research Assistant at University of Koblenz-Landau.
- 2022 Vrije Universiteit Brussel (VUB)

Contents

Chapter 1

Introduction

1.1 Mining Software Repositories

Important practical aspects of developers' everyday work manifest in software repositories [BRB+09, KGB+14]. Typical studies in MSR and ESE attempt to better understand software development by employing this data, in terms of mining software repositories [CIC16]. Using such existing data source for research is an alternative to experimenting in software development. Experimenting is often intrusive and expensive (see [MB20, JLY+19, TLPH95, SHH+05]).

Empirical studies in software engineering use software repositories as data sources to understand software development. In particular, research studies models, abstractions, variables, or metrics that reflect API migration [SPN+18, RvDV17, RvDV12, RvDV13], developers experience [RRC16, RD11, ETL11], software changes [MPS08, SZZ05, YMNC04, KYM06, MSR17], entropy of changes [Has09], infrastructure as code [OZR22, OZVR21], dependencies [SPN+18, HV15, OBL10, RvDV13], network metrics [ZN08], diversity [VPR+15], similarity [LKMZ12, APM04, CCP07, MMWB13, SL16, HAL18], architecture [LLN14], documentation [AHS14], source code [ETL11, CDR18, FOMM10, GKSD05, CCP07], static code attributes [MGF07], change bursts [NZZ+10, Cho20], corrective engineering, bugs, defects and fixes [MW00, MGF07, RD11, Has09, ZN08, MPS08, SZZ05, NZZ+10, FBF+20], commit time [ETL11, SZZ05], pull requests [GPvD14, BPWS20, GZSvD15, GSB16, YWF+15, TDH14], open-source collaboration [Cho20], branching [KPB18], tests [BFS+18], OO-metrics [BBM96], asserts [CDO+15, KL17], social factors [FBF+20, VPR+15] model-driven technologies [KMK+15, HHL18, RHC+19, RHH+17], project popularity [AHS14, BHV16, WL14], languages usage [BTL+13, SHL+19], software builds [MAH10, GdCZ19] or reviewer assignment [RRC16, SdLJPM18].

Repository data is either used to answer questions that guide the decision-making in the software development (e.g., [Has06, Has08, TH18, VPR+15]), or to provide tools that help with practical aspects of developers' everyday work (e.g., [ZN08, RRC16]). Studies are classified into the field of Empirical Software Engineering (ESE), and more specifically into Mining Software Repositories (MSR).

1.2 Technical and Methodological Challenges

Studies working with repository data often focus on their results. Results are statements or tools, derived from the data, that help with practical aspects of software development. This thesis focuses on the methods and high-order methods used to produce such results[1]. In particular, we focus on incremental methods to scale the processing of repositories, declarative methods to compose a heterogeneous analysis, and high-order methods used to reason about threats to methods operating on repositories. We summarize this as technical and methodological improvements, motivated by previous work facing the same, and a series of related challenges [BRB+09, KGB+14, KPB18, Has08, APPG19, DRNN14, DNRN15, SPN+18, TDH14, RvDV17, SJAH09, SAH10, RRH+20].

For instance, before authors can study results on library updates in [SPN+18], four weeks of data extraction are needed (scalability). To better understand versioning conventions in [RvDV17], authors need an aggregate analysis runtime of 5.5 years (scalability). Authors of [PZS+20] need three months on a 56-core server before they can study refactorings (scalability). In [RRH+20], Java heuristics are needed to understand the heterogeneous technology stack of MDE technology (heterogeneity). To study social and technical factors in the context of pull requests, authors of [TDH14] rely on a valid method of using AIC [Aka98]. AIC is needed to decide between structurally different models to prevent threats of under- and overfitting (validity). Such challenging scenarios require effective methods and high-order methods to work with repository data.

We contribute the improvements to methods and high-order methods in the context of MSR/ESE to produce future empirical results more effectively.

1.2.1 Technical Challenges

Two technical challenges are handled in this thesis. They relate to mining the repository for variables, metrics, or abstractions, collecting raw data from the repository. This is the first step for many empirical studies. We identify and handle two challenges, i.e., the *scalability* and *heterogeneity* challenge, and explain them in the following two paragraphs.

Scalability Challenge Typical studies may benefit from data on the full revision history of the software under version control [GHJ98, GKMS00, Has06]. To this end, we may need to compute abstractions of the repositories, that do not only reflect the most recent revision, but abstractions that reflect all or many revisions [KPB18, APPG19, LDKBJ22]. This can improve understanding of the software development process, e.g., if the upcoming analysis builds advanced models that abstract over the time, or the branching structure of the

[1] We use the term *high-order method* to describe methods that operate on methods. Such high-order methods may solve methodological challenges by the study of methods used in empirical research. While not explicitly talking about *high-order methods*, we can find the same distinction between methods and high-order methods in recent statistic work, like in Gelman et al. [GHV20], where simulations with fake data (the high-order method) are used to better understand how the models are fit (an alias for several methods central to statistic science).

repository. Studies including such data and advanced models can be found here [PFS+20, SPN+18, AHS14, KPB18, GdCZ19, VPR+15, FLHV22]. However, compared to computing abstractions of single revisions, computing abstractions of all or many revisions in a repository is expensive. If studying repositories with more than a few revisions, which is not unusual, we meet a scalability challenge (see [SJAH09, APPG19, LDKBJ22]).

Heterogeneity Challenge Typical studies that use repository data to understand software development need to understand a heterogeneous (or diverse) technology stack, too, that potentially manifests in a repository and its fragments [HK06, PML15, SBH+19]. For instance, empirical studies do not get rich abstractions of the source code for free (ASTs). Studies need to understand the code in its surrounding, including technological aspects, like the build system [MAH10, LPS11, GdCZ19], dependencies management [LPS11, SB15], various possible interoperating languages [BTL+13], infrastructure as code [OZR22, OZVR21], or MDE technology invoking code generation [ZS06]. Even when limiting following up analysis to a very specific mixture of technologies (e.g., to mobile apps [SPN+18] or graph query languages [SHL+19]), an initial understanding of the technology stack, used in a repository, is still needed as an inclusion/exclusion criteria. Defining abstractions over a heterogeneous technology stack can be complicated due to the flexibility of how technologies compose and interrelate (see work on documenting technologies and languages in software projects, which is motivated by such heterogeneity [FLV12, LV14, HLV17, HHL+17, Hei22, RRH+18, RRH+20]). Concrete analysis meets a challenge with such heterogeneous repositories.

1.2.2 Methodological Challenges

We define a methodology to be the set of methods used in a study. Methodological challenges refer to problems with this set of methods and can be solved by the study of methods. Methodological challenges are not necessarily related to concrete results. They are related to results of a method (or set of methods) in a broader sense. We handle one methodological challenge that we refer to as the *validity challenge*. Such challenge aims at making statements about the validity of methods under clear conditions. We refer to methods dedicated to the study of methods as high-order methods.

Validity Challenge A typical methodology in MSR/ESE does not interpret the raw abstractions, metrics, or variables, mined from a repository. Instead, it complements the raw variables, with unobserved or unobservable variables, which are assumed to be relevant to understand software development. This might be confounding variables, or interesting parameters, relevant to understand a relationship between other variables. To this end, studies use methods to build complex models of the software development, to infer (learn) the unobserved aspects. See studies that follow such practice [KSA+13, FBF+20, VPR+15, BHV16, FLHV22, SHL+19, MW00, YXF+20, JTH21, NZZ+10, RRC16, ZPZ07, ZN08, TMHI16, TDH14, TH18].

Assessing the validity of data analysis methodology, that aims to understand and use such variables, and typical relationships between variables, within the limits of

precisely defined threats, is often challenging. This motivates many books that help with data analysis methodology in general [CCWA13, McE20, IR15, Agg15, Har15, DB18, GH06, GHV20]. This literature might not immediately be applicable to MSE/ESE. For software engineering, in particular, a literature review of frequent statistic errors that may threaten a methodology can be found in [RDCJ18].

The methodological complexity of data analysis reaches its peak in understanding and using *causation*, crucial for every scientific domain. A general discussion of leading authors examining causation can be found in [IR15]. Other examples of concrete challenges that arise are the sampling process of such variables [MHJ+15, DAB21], controlling variables [TH18] or correlated variables [JTH21]. Such challenges complicate the methodology of MSR/ESE research in that the validity may sometimes not be clear and threats are hard to disambiguate.

1.3 Technical and Methodological Contributions

This thesis includes a subset of the contributions published in three papers. The critical mass of this thesis, however, cannot exactly be isolated to the three publications.
We will provide an exhaustive list of publications, done over the past years in our working group, with overlapping authorship, that is strongly relevant for the critical mass of this thesis. Our previous publications are relevant to this thesis for the following reasons:

- The **challenges and motivations** for this thesis, and the contributed improvements to (high-order) methods, can be found in our previous publications.
- The **(high-order) methods** presented in this thesis are developed in the background of the publications contained in the list and are already used in some of them silently.

The following list enumerates all publications, where titles in **bolt** are formally included in this thesis. A detailed overview of all publications, including their connections, is given in Chapter 2.

Ref.	Year	Included	Title
[HAL18]	2018		Classification of APIs by Hierarchical Clustering.
[RRH+18]	2018		Systematic Recovery of MDE Technology Usage
[HHL18]	2018	Chapter 4	**EMF Patterns of Usage on GitHub**
[SHL+19]	2019		Empirical Study on the Usage of Graph Query Languages in Open Source Java Projects
[HL20]	2020	Chapter 3	**Incremental Map-Reduce on Repository History**
[RRH+20]	2020		Understanding MDE projects: megamodels to the rescue for architecture recovery
[HHL20]	2020		Reproducible Construction of Interconnected Technology Models for EMF Code Generation
[HL22]	2022	Chapter 5	**Operationalizing Threats to MSR Studies by Simulation-Based Testing**
[HL23]	2023	Chapter 5	**Operationalizing Validity of Empirical Software Engineering Studies**

The publications included in this thesis and the corresponding contributions can be summarized as follows.

1.3.1 Incremental Map-Reduce on Repository History

Studies working with abstractions of repositories with a high revision count face a scalability challenge.

We propose a method to improve the scalability of functions that abstract over repositories with high revision count in a theoretically founded way. We use insights on abstract algebra and program incrementalization to define a core interface of high-order functions that compute scalable static abstractions of a repository with many revisions. Extended map-reduce primitives are built on top of the core interface so that actual users can compose complex abstractions using the primitives, without noticing the underlying technical efforts on incrementalization. We thereby provide the first scaling solution that uses incrementalization to mine repositories.

This stands in contrast to previous work, assuming that incrementalization is not applicable in MSR/ESE [SJAH09].

The mechanism that we present is generally applicable to circumvent bookkeeping, with is typically needed when manually applying program incrementalization. We illustrate and evaluate the improvements by a concrete empirical placeholder, using map-reduce primitives to compute and aggregate cyclomatic complexity metrics for the files part of a revision. Previous work did show the successful use of map-reduce in the context of MSR/ESE [SJAH09].

Incrementalization is orthogonal to other options to improve the scalability. We show this by also enabling the reduction of redundancies and distributed processing in our prototype. We thereby manage to better compare to existing competitor in MSR/ESE.

We evaluate the scalability of our method by benchmarks, comparing a prototype with available competitors in MSR/ESE. We compare with LISA [APPG19] that reduces redundancy and with DJ-Rex [SJAH09, SAH10] that migrates an analysis to a distributed map-reduce framework. Our prototype outperforms both in terms of the time needed to process a repository and uses less memory than LISA.

1.3.2 Repository Mining with Datalog

Studies working with abstractions of repositories with a very heterogeneous technology stack may face difficulties composing functions, due to the flexibility of how technologies compose and interrelate.

We propose a method to improve the definition of functions that abstract over a repository with a heterogeneous technology stack, by using concepts from declarative logic programming (in particular Datalog [DEGV01, GHLZ13]) and combining them with ideas on megamodeling and linguistic architecture (see [FLV12, LV14, HLV17, HHL$^+$17, Hei22, RRH$^+$18, RRH$^+$20]). We reproduce existing ideas on declarative logic programming with languages close to Datalog, coming from architecture recovery [MMW02, MT01, TM03], source code querying [HVdM06], and static program analysis [BS09, SBEV18], and transfer them from the analysis of a homogeneous to a heterogeneous technology stack. In particular, we facilitate understanding a complex technology stack in a repository by a bottom-up, step-by-step, and modular classification of the repository, its fragments, and the involved technologies, using Datalog rules. The method finally leads to a non-trivial understanding of the repository and the involved technologies in the large. Results conform to schemata from previous work on megamodeling and linguistic architecture, which are developed to document how complex technologies and languages manifest in software projects.

Our method facilitates modularity to fight the heterogeneity present in a technology stack. We use modular rules to infer classifications from other existing classifications. Rules produce classifications that conform to ideas on megamodeling and linguistic architecture, ideas that have proven to be well suited to describe the complex composition of technologies in previous work [FLV12, LV14, HLV17, HHL$^+$17, Hei22, RRH$^+$18, RRH$^+$20]. Starting with basic classifications, e.g., of the revision's resources, our method uses modular Datalog rules that trigger more complex classifications, e.g., of the build system, Java, XML, or MDE technology. Finally, we may apply overall complex classification of technology patterns which might be part of

the repository. Our classifications can be read like proof-derivations, which helps to trace interrelations between modular rules, separately classifying independent parts of technologies.

We provide a prove-of-concept of such method in a case study. We apply the previously reoccurring ideas from architecture recovery [MMW02, MT01, TM03], source code querying [HVdM06], and static program analysis [BS09, SBEV18], to a novel, heterogeneous context, studying the Eclipse Modeling Framework (EMF). EMF is a very heterogeneous technology combining XML, Java, OSGI and various build systems in its application [SBMP08]. This heterogeneous context requires a more flexible access to the repository and its fragments. The case study defines and runs the Datalog rules to mine EMF technology patterns. We step-by-step classify different artifacts, part of EMF, find relationships (code vs. model vs. generator), and finally do a high-level classification that detects specific EMF technology usage patterns. We separate the rules for all technologies, and finally apply rules that classify relationships. We apply the mining to GitHub repositories.

1.3.3 Simulation-Based Testing

We propose a high-order method to improve the disambiguation of threats to methods used in MSR/ESE. Data analysis methods, like regression modeling, statistic tests or correlation analysis, are in active use with the aim to understand and use software engineering data to improve software engineering practice. Assessing the validity of data analysis methodology in this context, within the limits of precisely defined threats, is challenging. We focus on a better disambiguation of threats, operationalizing reasoning about them, and making the implications to a valid data analysis methodology explicit, by using simulations. We encourage researchers to accomplish their work by implementing 'fake' simulations of their MSR/ESE scenarios, to operationalize relevant insights about alternative plausible results, negative results, potential threats and the used data analysis methodologies. The simulation replaces real data by 'fake' data, substituting observed and unobserved variables, related to a real scenario, with synthetic variables, carefully defined according to plausible (or controversial) assumptions on the scenario. The simulation allows to critically explore how the methodology reacts in such a transparent scenario. This is not possible on real data, since the reality is never fully transparent. A simulation thereby manifests as an artifact that can accomplish research by disambiguating its threats and their impact. We prove that such way of simulation-based testing contributes to the disambiguation of threats in published MSR/ESE research.

1.4 Summary of the Delta to the Publication

This thesis is based on three central publications.

- Chapter 3 (Incremental Map-Reduce on Repository History) is published in [HL20].
- Chapter 4 (Repository Mining with Datalog) is published in [HHL18]. This thesis includes the method proposed in [HHL18]. The empirical results com-

puted in the case study of [HHL18], using the method, are associated with Heinz [Hei22]. We discuss the case study as a proof-of-concept of our method.

- Chapter 5 (Simulation-Based Testing) is published in [HL22, HL23].

Other publications, that are strongly related but not directly included, are listed in Chapter 2. Details on the correspondence between publications and thesis will be discussed within each chapter. A detailed overview is given in Chapter 2.

1.5 Road-map of this Thesis

Chapter 2 starts with an overview of the contributions of this thesis, describing the interrelations and connecting it to a set of other publications done in the working group. The Chapters 3-5 cover the central contributions included in the thesis. Most parts of the Chapters 3-5 can be read isolated. We summarize limitations of all contributions in Chapter 7. The related work will be discussed in Chapter 8. For the conclusion and future work, see Chapter 9.

1.6 Metamodel for the Chapters 3-5

The Chapters 3-5 will follow the same metamodel:

- Each chapter will start with an introduction section.
- A motivation section reviews selected work and illustrates challenges that motivate the contribution.
- A background section summarizes established knowledge that backs the contribution. Some content in the background sections has never been discussed in the context of MSR/ESE before. We try to keep the background sections as close to the original contributions of this thesis as possible, not introducing any additional toy examples. Hence, the presentation in the background sections can already be considered as an original part of each chapter's contribution. The background part will be skipped in Chapter 5 as there is no previous work relevant for the contribution in the context of MSR/ESE.
- The main part of each chapter introduces the details of the contribution.
- Hereafter, the evaluation is presented.
- Finally, a conclusion for the individual chapter is presented.
- The related work is collectively discussed in Chapter 8.

Chapter 2

Overview

This chapter gives an overview of the composition of this thesis by the means of iterating through its development process. We will discuss the relevant publications.

2.1 Publication List

We start with a chronological list of all publications that can be found in the close context of this thesis. The publications can be found in Table 2.1. The publications are done by our working group, at the University of Koblenz-Landau, Germany, and in cooperation with a working groups from the University of L'Aquila, Italy.

Table 2.1 is composed as follows: The publications **in bolt** include the central contributions of this thesis. The authorship of three of them ([HL20], [HL22] and [HL23]) can entirely be associated with this thesis (and Johannes Härtel). Concrete empirical results of the case study published in [HHL18] (on the usage of EMF on GitHub) are associated with Marcel Heinz and central to his thesis (see [Hei22]). This thesis will sketch the case study, since we need it as a proof-of-concept for the method we proposed to mine a heterogeneous technology stack. The technical and methodological insights of using Datalog as a method for mining, are entirely associated with Johannes Härtel and this thesis. We will again point out this detail on the authorship in the remainder of this chapter and in Chapter 4.

The remaining publications of Table 2.1 are stepping stones, taken on the way to this final document. Not all, but most publications include major contributions attributed to other leading authors. The full list of publications is discussed in this chapter, to understand the broader context, the development of this thesis, and the relations among the publications. In this chapter, we will acknowledge how work builds on each other, how ideas evolve, get reused, and are made more mature over the time. This chapter will present the big picture.

2.2 Publication Delta

This thesis reuses passages of the original publications. We will improve the description of the contributions, but we will not apply fundamental change to the presented contributions. For the main publications ([HL20], [HL22, HL23] or [HHL18]), details on the delta between publication and thesis can be found in the corresponding

Table 2.1: Table of publications with contributions related and included in this thesis

Ref.	Year	Included	Title
[HAL18]	2018		Classification of APIs by Hierarchical Clustering.
[RRH+18]	2018		Systematic Recovery of MDE Technology Usage
[HHL18]	2018	Chapter 4	**EMF Patterns of Usage on GitHub**
[SHL+19]	2019		Empirical Study on the Usage of Graph Query Languages in Open Source Java Projects
[HL20]	2020	Chapter 3	**Incremental Map-Reduce on Repository History**
[RRH+20]	2020		Understanding MDE projects: megamodels to the rescue for architecture recovery
[HHL20]	2020		Reproducible Construction of Interconnected Technology Models for EMF Code Generation
[HL22]	2022	Chapter 5	**Operationalizing Threats to MSR Studies by Simulation-Based Testing**
[HL23]	2023	Chapter 5	**Operationalizing Validity of Empirical Software Engineering Studies**

chapters (Chapter 3, 4 and 5).

2.3 Contribution Types

The contributions included in this thesis are different from standard instances of empirical research in MSR/ESE. In this thesis, the concrete empirical questions and results, which are the usual contributions of empirical research, are taken as placeholders. We use such placeholders to illustrate improvements to methods and high-order methods used to do empirical research. We summarize this as *technical and methodological* improvements in MSR/ESE.

When looking at the publication list (Table 2.1), we will discuss different types of contributions. We distinguish between empirical contributions that focus on results, and technical/methodological contributions that focus on methods and high-order methods. The following sections give a brief description of the types as we use them in this thesis.

2.3.1 Empirical Contributions

Empirical contributions that use a repository as data source either answer questions that guide the decision-making in the software development (e.g., [Has06, Has08, TH18, VPR+15]), or provided tools that help with practical aspects of developer's everyday work (e.g., [ZN08, RRC16]). Take the following examples:

- In [VPR+15], authors focus on statements on the relation between gender diversity, tenure diversity and productivity. Using such results in decision-making may guide recruiting departments when composing teams.
- In [ZN08], models predict defects using network metrics. Such predictions can be used as a tool to help developers with their everyday work when trying to find defects.

A set of methods is involved in such studies to collect and analyze the data. We find regression modelling, computation of network metrics, correlation analysis, or the Blau index (see [Bla77] for the Blau index). Evolving this set of methods is not the central contributions of the previous publications.

2.3.2 Technical and Methodological Contributions

Evolving the methods used in such research is equally important. Works may examine a placeholder question by a set of methods, while contributing an improvement to the underlying set of methods and its understanding.

Take the example of [LSBV17] that examines cyclomatic complexity, and compare it to [APG17], that illustrates a new method to scalable MSR/ESE analysis by showing how it computes the cyclomatic complexity. Both works have a fundamentally different interest in why they compute cyclomatic complexity. For the latter, findings on the cyclomatic complexity, the concrete empirical results, are less relevant. The general technical improvement of the method matters.

Take another example of [TMHM17], that examines a set of model validation methods, and compare it to [TDH14] that examines social and technical factors while applying AIC [Aka98] as a method to protect against overfitting.

Ideally, a work that aims at answering a concrete empirical question puts less attention on reinventing methodological and technical aspects. Often, an established set of methods can be used.

If a publication focuses on improving methods and high-order methods, we further distinguish between technical and methodological contributions.

Technical Contributions

We consider technical contributions as those improving methods according to technical objectives, like scalability, or composability.

For instance, studies may need processing methods that are complicated to implement or hard to scale. For many empirical contributions, correct and efficient methods for data extraction and data storage are a mandatory prerequisite. Technical solutions typically apply some sort of code analysis to repositories and its fragments or store related content (e.g., [CJ18, TME+18, SPN+18, FBF+20, DRNN14, APPG19, SJAH09, BPVZ20]).

Several works in MSR contribute technically to the methods used in MSR/ESE (e.g., [DRNN14, APPG19, SJAH09, BPVZ20]). Often, such solutions try to facilitate the analysis of repositories and its fragments, without being limiting too much to concrete empirical questions. They pick up the most relevant technical characteristics of an MSR/ESE data collection method.

A strategy used in our and other work is to use existing software languages, or storage mechanisms, and tailor them to the needs of analyzing repositories. Signal/-Collect queries are run on repositories in [APPG19]; Map-reduce is used in our work and in [SJAH09]; domain-specific languages (DSLs) are examined in [DRNN14]; Datalog is examined in our work; graph compression for storage in [BPVZ20]. All solutions can be used as a technical building block, as methods, to answer a wide range of potential empirical questions.

Methodological Contributions

We define a methodology to be the set of methods used in a study. Methodological challenges refer to problems with this set of methods and can be solved by the study of methods. Methodological challenges are not necessarily related to concrete results. They are related to results of a method (or set of methods) in a broader sense. We handle one methodological challenge that we refer to as the *validity challenge*. Such challenge aims at making statements about the validity of methods under clear conditions. We refer to methods dedicated to the study of methods as high-order methods.

Methodological contributions may include examining the evaluation of methods, the applicability of quality criteria (like validity), or the meaning of threats in such context. Selecting the right set of methods to do an empirical study is still challenging and requires such high-order discussion.

We find related work that executes such high-order discussion of methods, e.g., methodology on how to build models (e.g., [TMHM17, TH18, JTH21]).

In [TMHM17], for instance, model validation techniques are reviewed. This thesis follows the same direction, contributing simulations as a high-order method, to better understand the methods involved in an empirical study and the corresponding results.

2.4 Publication Timeline

In the following, we iterate Table 2.1 in chronological order and discuss the empirical, technical and methodological contributions of the publications. We explain how the contributions relate to this thesis. We also discuss some hidden technical and methodological relations between the publication that have not been included in the original papers.

2.4.1 Classification of APIs by Hierarchical Clustering.

	Metadata
Authors	Johannes Härtel
	Hakan Aksu
	Ralf Lämmel
Venue	ICPC
Year	2018 (May)
Reference	[HAL18]

The evolution of this thesis starts with an unsupervised classification model for Application Programming Interfaces (APIs) published in [HAL18]. The classification is done by hierarchical clustering.

APIs are crucial to software development. Building a model for API classes helps developers to find the right API, e.g., by getting a list of available alternatives for a given API. This motivation is also picked up in more recent work on the topic (see [VCR22]). This early publication puts a clear focus on the concrete empirical result that helps developers to better use APIs. The work also provides a tool for doing this.

In particular, the model clusters APIs based on code identifiers. The raw data is mined from JARs that we get from Maven Central. The clustering groups APIs together that provide syntactically related functionality. The predicted clusters are compared to preexisting categories, like *XML*, *Security* or *Database*. We use two baselines of such categories, one from a previous work [RLP13], and one crowd-sourced classification from Maven Central.

Technical and methodological challenges to involved methods are partially included, but they are not the primary focus of this research.

This work includes a concrete technical challenge on scalability, which is to analyze a list of 2.5 million POM files from GitHub. We solve it using distributed map-reduce, which is not reported in our publication. The usage of map-reduce is motivated by previous work of Shang et al. [SAH10]. The successful application is one of the motivations for our future efforts in incrementalizing map-reduce for the application in MSR/ESE.

Methodologically challenging is how the publication builds the model for making predictions. The model is fitted using our baseline classifications. The model includes unknown parameters that decide on alternatives in the classification mechanism, in particular, how the code identifiers are extracted, NLP processing is applied, and hierarchical clustering works. Parameters are fitted (learned) by a grid-search going through all possible parameter combinations. Advanced algorithms for exploring the parameter space are not applicable due to its discrete nature. We describe the parameter space in terms of a feature model which is somehow novel.

However, we also face methodological weaknesses, that we have not been aware of at this time. Corresponding threats have raised the interest in the later development phase of this thesis. Most importantly, the presented work does not follow a methodology to prevent overfitting. We assume that future work on API clustering should be more careful about this threat if fitting parameters to existing data. Our work on simulation-based testing, which appears four years after this publication, is dedicated to making such threats more explicit.

2.4.2 Systematic Recovery of MDE Technology Usage

	Metadata
Authors	Juri Di Rocco
	Davide Di Ruscio
	Johannes Härtel
	Ludovico Iovino
	Ralf Lämmel
	Alfonso Pierantonio
Venue	ICMT
Year	2018 (June)
Reference	[RRH+18]

About the same time of doing our research on APIs, our working group, and a working group from University of L'Aquila, started cooperating on the examination of *model-driven engineering* (MDE).

MDE [Béz05b, Béz05a, Sch06] has the goal to increase the quality and productivity in software development by the usage of models, metamodels, model transformations, and model comparisons (note the ambiguity of models in this thesis). In essence, developers try to work with (high-level) models instead of low-level code (also considered as model) whenever possible. Often this is done to abstract from implementation-specific details, that are finally derived from the high-level models using model transformations. Models may also be used in the communication between developers, like often done using UML. Adoption of MDE in software development is still subject to empirical research and the benefits are discussed controversially, e.g., see work on EMF related technology in [KMK+15], or work on the relation between UML and software quality (defects) examined in [RHC+19, RHH+17].

To gather raw data on MDE practice, our first publication on this topic proposes a method to extract the usage of ATL transformations from open-source repositories. This work is published in [RRH+18].

Mining of MDE technology is technically challenging. MDE technology is often very specific, project content is heterogeneous, and the analysis requires non-standard methods to understand the different artifact types and how they relate to each other. The method that we present in [RRH+18] uses *heuristics* to produce a graph of connected MDE artifacts located in a subject MDE repository. A heuristic is implemented in plain Java and queries the file system. The implementation of the heuristics is very flexible since Java is a general purpose programming language.

However, during the development of the method, we noticed that such flexibility weakens guarantees that we can give about the heuristics and their execution, e.g., on modularity and complex interrelationships between them. Implementing heuristics within a more constrained programming language, with stronger guarantees, but that still allows mining of the complex artifact types, is the motivation for our following up efforts using declarative logic programming with Datalog (see next section).

The work published in [RRH+18] focuses on the technical aspects of recovering MDE artifacts and relationships. The technical aspects of our recovery method are presented in depth and evaluated for precision and recall, manually tagging the expected results.

This publication does not include strong empirical contributions. No advanced models are built that help to better understand the software development using MDE techniques. Further, there is no representative sample of software projects used, since the MDE projects studied come from a curated suite of ATL projects (ATL Zoo). Such collection is likely to be different from ATL usage in the wild. However, our publication argues the contributed method to collect raw data on the relationships between MDE artifacts is still helpful for developers in the concrete cases, since it can be used to understand the project while onboarding.

A technical insight of this work is that we spotted the first instance of a reoccurring technical problem of methods that access the content of repository revisions through the operating system's file layer. This indirection typically turns out to be a performance bottleneck, especially if analyzing multiple revisions of the same repository. We find related discussions in publications like [APPG19]. In the upcoming publications of Table 2.1, we solved these issues using virtualized access by dedicated primitives that immediately map file system access to the repository's object storage (using git bare repositories).

2.4.3 EMF Patterns of Usage on GitHub

	Metadata
Authors	Johannes Härtel
	Marcel Heinz
	Ralf Lämmel
Venue	ECMFA
Year	2018 (June)
Reference	[HHL18]

A second branch of concurrent research did examine the implications of a data analysis method with stronger limitations in the programming language, when com-

pared to the previous weakly constrained usage of general purpose Java. Such efforts also aim at the technical challenge of mining heterogeneous MDE technology. The work is published in [HHL18].

In the publication [HHL18], the empirical placeholder is EMF technology. ATL and EMF are very related. Both are well-known instances of MDE technology. One benefit of studying EMF is that the gathered results on EMF usage is relevant to Heinz [Hei22]. The results conform to a pattern catalog developed by Heinz that he reports on in the context of a thesis on knowledge engineering. In the publication [HHL18], but especially in this thesis, we contribute the method used for mining the heterogeneous technology stack of EMF.

We propose a method to improve the definition of functions that abstract over a repository with a heterogeneous technology stack, by using concepts from declarative logic programming and combining them with ideas on megamodeling and linguistic architecture. We reproduce existing ideas on declarative logic programming with languages close to Datalog, coming from architecture recovery, source code querying, and static program analysis, and transfer them from the analysis of a homogeneous to a heterogeneous technology stack. We provide a prove-of-concept of such method in a case study.

The method is discussed in detail in Chapter 4. However, there are small differences between the publication and the thesis. In the original publication, we use Apache Jena (a Datalog dialect). Occasionally, our specialize language for mining is called *QegaL* due to its original purpose of querying linguistic architecture. In this thesis, we change the presentation to standard Datalog and improve the related work discussion on declarative programming. We applied this change to get a more standardized discussion that better aligns with the related work.

Furthermore, future work on incrementalization is possible with Datalog. Incrementalization was already planned to be part of our future work, but after publishing the Datalog method, we decided to start with incrementalizing map-reduce as the more established language for data processing and mining (see [SAH10]). Incrementalizing map-reduce is part of the upcoming publications discussed in the context of Table 2.1. It is the central topic of Chapter 3.

2.4.4 Empirical Study on the Usage of Graph Query Languages in Open Source Java Projects

	Metadata
Authors	Philipp Seifer
	Johannes Härtel
	Martin Leinberger
	Ralf Lämmel
	Steffen Staab
Venue	SLE
Year	2019 (October)
Reference	[SHL+19]

The next publication focuses on concrete empirical results of examining the usage of graph query languages on GitHub. The paper is published in the Software Lan-

guage Engineering (SLE) community (in [SHL+19]), which shares a natural interest in empirical data on the usage of (graph query) languages in the wild.

The paper compares the evolution of the usage of available graph query languages. The technically challenging analysis of the full revision history is limited to SPARQL and Cypher. To get a detailed picture of the usage of both languages, we first identify potential occurrences in repositories by analyzing project dependencies. Afterward, we apply heuristics that indicate if a piece of code includes use of SPARQL and Cypher.

At this time, a predecessor of our Topleet prototype (incremental map-reduce) was already available, so we used map-reduce code to run the heuristic. Finally, we examine a set of 7274 repositories and over one million revisions. We did not report on this since our incremental map-reduce method was not yet published. The successful application was another motivation for us to keep up with using map-reduce as a method for the incremental processing of revisions.

Other than that, the publication relies on very basic methods to report and compare the raw usage data. To some extends, this raw data is interesting since we render it over more than ten years. Our paper informally explains the evolution of the languages by characteristic events (like new versions of a graph query language). At this point, we avoid advanced temporal models, which might have been part of a more formal set of methods to study such raw data.

The paper includes a very basic regression model. This model tries to capture the decision of a project for a query language, i.e., if it decides for SPARQL or Cypher. The methodology to do this again shows some flaws, for instance, we selected an inappropriate distribution for the output variable. Such flaws are part of our upcoming work on simulations. Simulations that show the implications of the wrong distribution form are already included in the online resources of [HL22], and might appear in an extended journal version of the publication on simulation.

2.4.5 Understanding MDE projects: megamodels to the rescue for architecture recovery

	Metadata
Authors	Juri Di Rocco
	Davide Di Ruscio
	Johannes Härtel
	Ludovico Iovino
	Ralf Lämmel
	Alfonso Pierantonio
Venue	Software and Systems Modeling
Year	2020 (January)
Reference	[RRH+20]

The publication [RRH+20] is a journal version of the previous work, described in [RRH+18]. The journal version extends the list of heuristic, e.g., adding the analysis of Acceleo MDE technology. This decreases the number of unrelated artifacts in the subject MDE projects, to better understand the relationships between the

different artifact types. Like in the previous conference version, the contribution of this journal version is an improvement to the technical method for collecting data.

The work makes the motivation for the proposed methods more explicit by stating that: '*diversity of employed languages and technologies blurs the picture making it difficult to analyze existing MDE-based projects*' (direct citation [RRH+20]). The work also emphasizes the usage of megamodeling as an underlying schema for structuring the extracted data.

2.4.6 Incremental Map-Reduce on Repository History

	Metadata
Authors	Johannes Härtel
	Ralf Lämmel
Venue	SANER
Year	2020 (February)
Reference	[HL20]

In February 2020, the method of using incremental map-reduce to process repository history was published, which is a technical contribution of this thesis. The publication is motivated by the successful application of map-reduce in the previous publications, and by the scalability problems that we repeatedly faced doing concrete empirical contributions.

We propose a method to improve the scalability of functions that abstract over repositories with high revision count in a theoretically founded way. We use insights on abstract algebra and program incrementalization to define a core interface of high-order functions that compute scalable static abstractions of a repository with many revisions. We evaluate the scalability of our method by benchmarks, comparing a prototype with available competitors in MSR/ESE.

This publication is one of the central publications included in this thesis. Details of this publication will be covered in depth in Chapter 3 of this thesis.

2.4.7 Reproducible Construction of Interconnected Technology Models for EMF Code Generation

	Metadata
Authors	Marcel Heinz
	Johannes Härtel
	Ralf Lämmel
Venue	JOT
Year	2020 (July)
Reference	[HHL20]

A work on the reproducible construction of interconnected technology models is one of the following up works. It technically relies on our Datalog method used to mine an EMF repository with a heterogeneous technology stack. The publication is central to the thesis of Marcel Heinz (see [Hei22]). To some extends, this work shows a more recent application of the Datalog method, which provides another

proof of the applicability. Methodologically challenging aspects of reproducibility can be found in this publication too.

2.4.8 Operationalizing Threats to MSR Studies by Simulation-Based Testing

	Metadata
Authors	Johannes Härtel
	Ralf Lämmel
Venue	MSR
Year	2022 (May)
Reference	[HL22]

While most of the previous publications did not involve complex methodology for building models of software development, unpublished efforts of us did.

In the first years of such efforts, we repeatedly consulted textbooks on modeling methodology, trying to adapt recipes (typically coming from other domains). However, we did not have much success with this strategy since the software repository data turned out to be too different from anything else.

A breakthrough came with the question of how to test if our methodology (the set of methods we use) works as expected. Motivated by very recent literature on statistic modeling by McElreath [McE20] and Gelman et al. [GHV20], we started using simulations. We add a direct citation of the first three sentences of the preface of [GHV20] to emphasize this trend: *"Existing textbooks on regression typically have some mix of cookbook instruction and mathematical derivation. We wrote this book because we saw a new way forward, focusing on understanding regression models, applying them to real problems, and using simulations with fake data to understand how the models are fit."* (Direct citation of Gelman et al. [GHV20])

In such recent work, models that are the central entities of any methodology to understand real data, are systematically complemented by models that are run on simulated 'fake' data. Such practice allows making objective statements on whether the model and methodology work as expected. Transferring such practice to MSR/ESE requires simulating repositories and software engineering data. We noticed that this is more systematic than copying established methods from unrelated domains.

The idea evolved and did appear in May 2022 [HL22]. We consider this contribution as a high-order method because it aims at making statements about the methodology (a set of methods), and does not aim at concrete empirical results.

We propose a high-order method to improve the disambiguation of threats to methods used in MSR/ESE. We focus on a better disambiguation of threats, operationalizing reasoning about them, and making the implications to a valid data analysis methodology explicit, by using simulations. We encourage researchers to accomplish their work by implementing 'fake' simulations of their MSR/ESE scenarios, to operationalize relevant insights about alternative plausible results, negative results, potential threats and the used data analysis methodologies. We prove that such way of simulation-based testing contributes to the disambiguation of threats in published MSR/ESE research.

We will describe the details in Chapter 5.

2.4.9 Operationalizing Validity of Empirical Software Engineering Studies

	Metadata
Authors	Johannes Härtel
	Ralf Lämmel
Venue	EMSE
Year	2023
Reference	[HL23]

A corresponding journal version was published hereafter.

Chapter 3

Incremental Map-Reduce on Repository History

3.1 Introduction

Empirical contributions in MSR/ESE often rely on data mined from repositories and its fragments (i.e., revisions, parent-relations, commit-metadata, or resources). Following up analysis typically depends on a process of abstracting from such raw repository data, using abstractions that are well-suited to measure relevant aspects of the software development.

Abstraction or metrics (often also called variable when used in following up models) may reflect API migration [SPN+18, RvDV17, RvDV12, RvDV13], developers experience [RRC16, RD11, ETL11], software changes [MPS08, SZZ05, YMNC04, KYM06, MSR17], entropy of changes [Has09], infrastructure as code [OZR22, OZVR21], dependencies [SPN+18, HV15, OBL10, RvDV13], network metrics [ZN08], diversity [VPR+15], similarity [LKMZ12, APM04, CCP07, MMWB13, SL16, HAL18], architecture [LLN14], documentation [AHS14], source code [ETL11, CDR18, FOMM10, GKSD05, CCP07], static code attributes [MGF07], change bursts [NZZ+10, Cho20], corrective engineering, bugs, defects and fixes [MW00, MGF07, RD11, Has09, ZN08, MPS08, SZZ05, NZZ+10, FBF+20], commit time [ETL11, SZZ05], pull requests [GPvD14, BPWS20, GZSvD15, GSB16, YWF+15, TDH14], open-source collaboration [Cho20], branching [KPB18], tests [BFS+18], OO-metrics [BBM96], asserts [CDO+15, KL17], social factors [FBF+20, VPR+15] model-driven technologies [KMK+15, HHL18, RHC+19, RHH+17], project popularity [AHS14, BHV16, WL14], languages usage [BTL+13, SHL+19], software builds [MAH10, GdCZ19] or reviewer assignment [RRC16, SdLJPM18].

Abstractions may strongly differ in how they measure the static or dynamic aspects of the software development that manifest in a repository. When having a function that abstracts over the repository, we can distinguish between[1]:

- **Static Abstractions**: We may abstract over single revisions only. Such static

[1]There might be different terminology describing such difference, like static vs. *change, process* or *churn*. We do not refer to the difference between static and dynamic program analysis. For examples, see work on defects [MPS08, DLR10]. We stick to static and dynamic.

abstractions do not reflect the dynamics of the repository. We can compute them as functions applied to a single revision and the resources part of it. For instance, we may compute the abstractions mentioned above *statically*, e.g., the total lines of code, the number of dependencies, or the presence of a given architecture pattern (in a given revision).

- **Dynamic Abstractions**: We may also abstract over more than a single revision. Such dynamic abstractions allow abstracting over differences between revisions, over a revision's past, over its future, or over a given time window relative to it. We can compute them as functions applied to sets of revisions and the resources part of it. For instance, we may compute the abstractions mentioned above *dynamically*, e.g., aggregates of past lines changed, future defects, or the entropy of the repository content in a time window (computed on a set of revisions).

Such abstractions may reuse each other. We may find dynamic abstraction built on top of static abstractions, e.g., the change (dynamic abstraction) of the total lines of code (static abstraction). We may find static abstraction built on top of dynamic abstractions, e.g., the total number (static abstraction) of previous changes to files (dynamic abstraction). Related work that systematically explores various forms of such abstractions, e.g., in the context of modeling defects, can be found here [MPS08, Has09, DLR10].

What static and dynamic abstractions have in common is that we always associate them with a single revision, or a point in time, where the abstraction reflects the software development process best. This is necessary for following up models that use abstractions as variables where the point in time, and the temporal precedence of variables, is typically relevant for claims on causation. See examples of work using various abstractions to build models [KSA+13, FBF+20, VPR+15, BHV16, FLHV22, SHL+19, MW00, YXF+20, JTH21, NZZ+10, RRC16, ZPZ07, ZN08, TMHI16, TDH14, TH18]. In particular, a static abstraction is always associated with a specific revision; a dynamic abstraction considers the past and future of, or a time window relative to, a specific revision.

The previous aspects on computing abstraction of revisions bring up a fundamental problem with the scalable computation of such abstractions when done for the repository's full revision history.

Typical studies may benefit from data on the full revision history of the software under version control [GHJ98, GKMS00, Has06]. To this end, we may need to compute abstractions of the repositories, that do not only reflect the most recent revision, but abstractions that reflect all or many revisions [KPB18, APPG19, LDKBJ22]. This can improve understanding of the software development process, e.g., if the upcoming analysis builds advanced models that abstract over the time, or the branching structure of the repository. Studies including such data and advanced models can be found here [PFS+20, SPN+18, AHS14, KPB18, GdCZ19, VPR+15, FLHV22]. However, compared to computing abstractions of single revisions, computing abstractions of all or many revisions in a repository is expensive. If studying repositories with more than a few revisions, which is not unusual, we meet a scalability challenge (see [SJAH09, APPG19, LDKBJ22]).

In this chapter, we will focus on the technical challenge to compute static abstraction of single revisions, but for all or many revisions of the repository (comparable to [APPG19, LDKBJ22]). We show how to scale such computations using methods to incrementalization. We limit us to static abstractions because it is most comprehensible and good for illustration. Our method may also be able to accelerate the computation of dynamic abstractions. We will present some ideas for future work and more details in this chapter and in the chapter on limitations (Chapter 7). However, dynamic abstractions are not part of our contribution, since we have not yet done any evaluations.

3.1.1 The Scalability Challenge

In practice, work that computes static abstractions for multiple revisions faces a scalability challenge. In the following enumeration, we list conceptual solutions for realizing scaling computation:

- Scalability can be reached by the method of *parallel and distributed analysis* [SJAH09, SAH10, DNRN15, DNRN13, NDNR14]. Work may rely on code written in map-reduce [SJAH09, SAH10] or domain-specific languages [DNRN15, DNRN13, NDNR14]. However, such work scales at the expense of high hardware usage.

- Some work reports on handmade *incrementalization* to accelerate the processing [CJ18, TME+18]. The typical analysis code calls *git-diff* and just examines the changed code lines between revisions. Such incremental analysis may be complicated because it requires additional bookkeeping efforts. The authors of [SJAH09] state that this is not applicable when prototyping (see the related work discussion of [SJAH09]).

- Another method is to *reduce redundancies and memoize intermediate results*, leveraging the high similarity between revisions. Such method is presented in [APPG19] and requires analysis to be written in Signal/Collect [SBC10].

- Other work relies on the method of *sampling* [MHJ+15, DAB21]. Here, a random subset of the target population, e.g., a sample of revisions, is examined. Results on the sample need to be annotated with estimates of uncertainty. Such work looses data.

- *Query optimization* is another method, applicable to improve the scalable processing. An intuitive example is to optimize the order of query steps, like filter and join. Filtering can often be executed before joining, which reduces the data in advance of expensive joining. Other optimizations, like changing the order of joints, may be applied based on runtime data. Recent frameworks, like Catalyst for Spark DataFrames [AXL+15], enable such optimizations.

The previous technical methods can be considered as orthogonal. They can be combined to improve the scalability of an MSR/ESE analysis which is tailored to the processing of single or multiple revisions. Incrementalization is specific to analyzing multiple revisions.

This chapter focuses on incrementalization. We exclude optimization by sampling revisions because this practice just reduces the size of the actual problem. We also do not include query optimization methods that may be run on top of our method; a fact that calls for future work.

We propose a method to improve the scalability of functions that abstract over repositories with high revision count in a theoretically founded way. We use insights on abstract algebra and program incrementalization to define a core interface of high-order functions that compute scalable static abstractions of a repository with many revisions. Extended map-reduce primitives are built on top of the core interface so that actual users can compose complex abstractions using the primitives, without noticing the underlying technical efforts on incrementalization. We thereby provide the first scaling solution that uses incrementalization to mine repositories. This stands in contrast to previous work, assuming that incrementalization is not applicable in MSR/ESE [SJAH09].

The mechanism that we present is generally applicable to circumvent bookkeeping, with is typically needed when manually applying program incrementalization. We illustrate and evaluate the improvements by a concrete empirical placeholder, using map-reduce primitives to compute and aggregate cyclomatic complexity metrics for the files part of a revision. We selected this placeholder because concrete empirical studies examine the cyclomatic complexity too, like [JMF14, LSBV17], and one of the direct competitor for the technical analysis of repositories, LISA, also illustrates its method taking cyclomatic complexity as a placeholder (see [APG17]). In an evaluation, we compare performance of incremental map-reduce with the direct competitors. Previous work did show the successful use of map-reduce in the context of MSR/ESE [SJAH09].

Incrementalization is orthogonal to other options to improve the scalability. We show this by also enabling the reduction of redundancies and distributed processing in our prototype. We thereby manage to better compare to existing competitor in MSR/ESE.

Our work is inspired by incrementalization theory to define the high-order functions forming our core interface. This includes insights on algebraic structures to define changes, and insights on (self-maintained) derivatives or homomorphisms, to define applied functions. General work using such insights are, e.g., [GGMS97, CGRO14, BROL14].

We evaluate the scalability of our method by benchmarks, comparing a prototype with available competitors in MSR/ESE. We compare with LISA [APPG19] that reduces redundancy and with DJ-Rex [SJAH09, SAH10] that migrates an analysis to a distributed map-reduce framework. Our prototype outperforms both in terms of the time needed to process a repository and uses less memory than LISA.

3.1.2 The Topleet Solution

The idea of incrementalization is to process changes [RR93]. Assume that we already know the input and output of a function. If the input to the function changes, incremental processing favors updating the output, based on the input's change, rather than recomputing the new output from scratch.

We need another function (a derivative) that maps the input change to the output

change. This mechanism may be less expensive, but it only works if we already have the previous input and output computed. The concept of incrementalization is known for a long time [RR93].

Now, consider a function that takes a revision and returns a static abstraction of it. If we want to compute the abstraction for two succeeding revisions (part of a repository's history), and already have the abstraction of the first revision, we would favor using a derivative to update the previous abstraction based on the changes between the two revisions. Take a trivial example abstraction, like counting the number of classes. Here, we can react to a new file with a new class in a next revision, by just adding +1 to our class count.

We know that, between succeeding revisions, often just a few resources change. In the best case, we can reuse a big part of the previous abstraction, which is the key insight of transferring incrementalization to the revision history.

Manually writing these derivatives, which are necessary for incrementalization, is cumbersome and error-prone. To avoid handmade solutions, general program and database incrementalization methods exist that take over the important bookkeeping in the background [GGMS97, CGRO14, BROL14]. Such solutions rely on theoretic insight on algebraic structures, derivatives and self-maintained derivatives (group homomorphisms). **We use the foundations and present the first method that transfers such practice to mining software repository revisions.**

We provide a well-motivated core interface to implement incremental functionality in terms of derivatives. The core interface delegates the function application to all revisions of a repository. We introduce a prototypical implementation called Topleet[2]. It implements the core interface, but it also provides the concrete derivatives for extended map-reduce functionality, build on top of the core interface. A user can finally compose complex processing in regular map-reduce syntax, without manually implementing derivatives.

Topleet analysis is embedded into regular Scala code and provides a syntax that resembles non-optimized map-reduce code, widely understood and used for analysis in MSR/ESE [SJAH09, SAH10]. We also add optimization by reducing redundancies and distribution to better compare to recent prototypes used in MSR/ESE.

We compare the scalability of Topleet to a method relying on a manual migration of code to distributed map-reduce [SJAH09, SAH10], a corresponding hand-made incrementalization, and to a reduction of redundancies [APPG19].

3.1.3 Summary of this Chapter's Contributions

- An adaptation of incrementalization theory for processing multiple revisions of the repository history.
- Topleet; a prototype for the scalable processing of multiple revisions in map-reduce style, employing generally applicable incrementalization mechanisms, reduction of redundancies and distribution.

[2]**Implementation, advanced examples, usage guidelines and supplementing evaluation data** are available under `http://github.com/topleet/topleet`.

- An evaluation comparing the performance of Topleet with LISA (reduction of redundancies) and DJ-Rex (distribution) and an evaluation of Topleet's infrastructure.

3.1.4 Summary of the Delta to the Publication

Parts of the following text are taken from the previous publication [HL20]. The following items describe the delta of this thesis to the publication:

- We add a discussion of alternative algebraic structures and their implications on practical solutions. This discussion is part of a new background section.
- We discuss a handmade incrementalization in detail, introducing the running example on a function aggregating cyclomatic complexity metrics for a revision.
- This thesis improves the distinction between general, repository specific and prototype-specific insights on incrementalization. We discuss general insights in the background section, while repository specific and prototype-specific insights follow in Sec. 3.4.
- The terminology changes towards using *derivatives* and *self-maintained derivatives* (instead of index-based application and homomorphism). We assume that this naming better aligns with existing work on incrementalization (e.g., [CGRO14]).
- We add a more formal discussion of properties of our proposed method and their correctness, which has not explicitly been given in the original publication. The original publication assures correctness by differential testing only.

3.1.5 Micro road-map of this Chapter

This chapter follows the metamodel defined in the introduction (Sec. 1.6). In particular,

Sec. 3.2 begins with a motivation, describing a running example on a function aggregating cyclomatic complexity metrics for a revision. This thesis uses the example throughout the chapter. Furthermore, we describe existing solutions coming from the related work.

Sec. 3.3 introduces the background on incrementalization. To avoid additional toy examples, the background section illustrates general incrementalization solving the running example. The term 'background' is used in a modest sense, as this section already contains content that has not been discussed in the context of MSR/ESE before.

Sec. 3.4 transfers the background on incrementalization to mining software repository revisions. The section also discusses implementation details of our prototype.

Sec. 3.5 evaluates our prototype by comparing different features of it with two related methods (DJ-Rex and LISA).

3.2 Motivation

We motivate this part of the thesis by an application-oriented view on recent solutions to processing multiple revisions, part of a repository's history. As the running example, we choose to compute and sum up McCabe's cyclomatic complexity on all Java files of a revision. The computation of cyclomatic complexity has been used for the technical presentation of LISA [AG15, APG17, APPG19], it fits into the Boa infrastructure [DNRN15, DNRN13, NDNR14] and can be migrated to a distributed map-reduces frameworks in analogy to DJ-Rex [SJAH09, SAH10]. The cyclomatic complexity has been subject to empirical studies in software engineering, e.g., in [JMF14, LSBV17, TH18].

Later in the background section (Sec. 3.3), we start developing an incremental solution for mining the sum of cyclomatic complexity on all Java files of a revision.

The running example on cyclomatic complexity is just a placeholder. Our final prototype allows using the full range of extended map-reduce functionality. We will give a concise summary of the limitations of our method later in Chapter 7.

3.2.1 Migration to Distributed Map-Reduce (DJ-Rex)

DJ-Rex is presented in [SJAH09, SAH10] as a proof-of-concept showing the migration of an existing MSR analysis to distributed map-reduce. We reproduce two solutions, according to the presentation of DJ-Rex, migrating a function that computes McCabe's cyclomatic complexity to a distributed map-reduce framework. We avoid to show aggregation here. We call the solutions 'DJ-Rex' and 'DJ-Rex Incremental'.

- The first solution, DJ-Rex, follows the conceptual idea of [SJAH09, SAH10] in a straightforward manner, just migrating a non-incremental solution to a distributed map-reduced framework. Hence, this solution only benefits from distribution.
- We also implement a second solution, DJ-Rex Incremental, migrating a handmade incremental solution to map-reduce. This solution benefits from distribution and incrementalization. We omit discussing DJ-Rex Incremental in this thesis, since we discuss a closely related solution in Sec. 3.3.1. However, this incrementalization is done 'manually' and invokes additional bookkeeping that we need to do as a user.

The most relevant code of the DJ-Rex solution can be found in Listing 3.1.

```
1 val resources: RDD[(SHA, Path, Resource)] = . . .
2 val mcCabe: RDD[(SHA, Int)] = resources
3     .filter { case (_, path, _) => path.endsWith(".java") }
4     .map { case (sha, path, resource) =>
5         (sha, computeMCC(resource))}
```

Listing 3.1: DJ-Rex Solution: An excerpt of our reproduction of [SJAH09] migrating `computeMCC` to distributed map-reduce.

In particular, we use Scala and Apache Spark for distribution. The Resilient Distributed Datasets (RDDs) [ZCD+12] provide a distributed collection of elements

that can be operated on in parallel. The operation `filter` restricts the resources on Java files and `map` calls the foreign function `computeMCC` to compute the cyclomatic complexity on resources. We thereby executed the processing in parallel and distributed manner.

The problem with such simple code is that, already for a medium-sized repository, like libgdx/libgdx[3], with around 14.000 revisions, it needs approximately seven hours on a single machine. Handmade incrementalization efforts optimizing this map-reduce code to just handle changed resources ('DJ-Rex Incremental') leads to a much faster solution, taking six minutes. We will give an introduction to the idea of such handmade incrementalization in the background section (Sec. 3.3.1).

However, handmade incrementalization invokes additional bookkeeping efforts and is error-prone. Avoiding such bookkeeping in a generally applicable incrementalization framework will be the central topic of this chapter.

3.2.2 Domain-Specific Languages (Boa)

Another method that can be used to analyze repository history on a distributed map-reduce platform is Boa [DNRN15, DNRN13, NDNR14]. To this end, the analysis is written in a domain-specific language (DSL). The following excerpt can be found in the reference documentation of BOA[4] answering the question: *'How many fixing revisions added null checks?'*. We show the original solution, which is focused on computing an abstraction of change between succeeding revisions. However, we assume that an adaptation that computes the cyclomatic complexity as a static abstraction over a revision is straightforward.

```
1  before node: ChangedFile -> {
2      // if this is a fixing revision and there was a previous version of the file
3      if (isfixing && haskey(files, node.name)) {
4          // count how many null checks were previously in the file
5          count = 0;
6          visit(getast(files[node.name]));
7          last := count;
8          // count how many null checks are currently in the file
9          count = 0;
10         visit(getast(node));
11         // if there are more null checks, output
12         if (count > last)
13             AddedNullCheck << 1;
14     }
15     if (node.change == ChangeKind.DELETED)
16         remove(files, node.name);
17     else
18         files[node.name] = node;
19     stop;
```

Listing 3.2: Boa Solution: 'How many fixing revisions added null checks?' (Adapted copy from the Boa reference documentation)

To determine if null checks have been added, the code counts null checks on the previous revision of a changed file (if there is one) and on the current version.

[3] `http://github.com/libgdx/libgdx`
[4] `http://boa.cs.iastate.edu/docs/index.php`

Comparable to the DJ-Rex Incremental solution, Boa code processes changes manually – eye-catching by the usage of types such as `ChangedFile` and `ChangeKind`.

Boa provides a web-based interface to the proprietary Boa infrastructure [DNRN13]. We do not report on its performance as we cannot reproduce this setup. However, we assume that the performance does not differ from the DJ-Rex solutions; we also assume that there is the same performance gap between change-oriented (incremental) and non-change-oriented (non-incremental) treatment. Again, we *manually* implement the incremental behavior.

3.2.3 Reduction of Redundancies (LISA)

LISA [AG15, APG17, APPG19] is a solution that reduces the redundancies in the multi-revision code analysis, and the first in this presentation that does not require handmade bookkeeping on how to handle changes.

LISA needs a registered parser for a target file extension and analysis code written in Signal/Collect [SBC10]. The code for the computation of McCabe's cyclomatic complexity can be found in the publications [APG17, APPG19]. LISA does not allow direct calls to foreign functions.

The computation of McCabe's cyclomatic complexity on the same repository requires 12 minutes; however, LISA has the highest memory footprint with 6.4 GB.

After discussing the competitors of our method, where some already involve manual incrementalization, we switch to the formal background on generally applicable incrementalization mechanisms.

3.3 Background

This section starts with the formal idea of processing changes by derivatives, which is the essence of program incrementalization. The idea will be capable to express the handmade incrementalization practice on repository revisions, described in the previous section, generally, applicable in terms of a core interface, and map-reduce functions build on top of it.

Typical formalization expresses incremental processing in terms of abstract algebra. This has been done for a while (e.g., [GGMS97, CGRO14, BROL14]). The formalization helps to understand and plug the generic parts of an incrementalization (rather than implementing all parts of a concrete solution manually). Many bookkeeping efforts can be circumvented.

In this section, we give an introduction to the most relevant parts, in the context of our running example. The main contributions will follow in Sec. 3.4, transferring the ideas to the processing of repository revision history.

3.3.1 Handmade Incrementalization

We begin with a handmade program incrementalization, manually turning a non-incremental solution into an incremental. This translation requires additional bookkeeping efforts that tailors the solution towards handling changes by manually implementing the derivative.

> Our running example corresponds to a function that **computes a static abstraction of a single revision**. The function aggregates the sum of the cyclomatic complexity of all Java files (part of the revision). The input type of the function is a bag of resources. The output is of type integer. **We apply this function to all revisions, part of the repository, to get a corresponding abstraction (integer) for each revision.**

Non-incremental Solution A non-incremental Scala solution, applying the function to each revision, can be found in Listing 3.3.

The solution uses a virtualized access to the resources, contained in the revision, by a mutable bag of resources. It uses two methods, `add` and `remove`, which inform the program of changing path-resources tuples, updating the bag accordingly. Add and remove is called while traversing the revision sequence (eventually backed by the object storage of the bare repository). We hide this unessential aspect of the code.

Such solution corresponds to methods using a working copy, located on the file-system, and successively checking out revision after revision. The solution presented here virtualizes such access using the bag of resources since an indirection over the file-system introduces enormous overhead. We simplify and assume that we have a sequence of revisions ordered by time (no branching) that we can traverse. In Sec. 3.4, we will generalize this practice to graph structures that are better suited to represent repository revision history.

The function `abstraction` reflects the function to compute the abstraction. It aggregates the cyclomatic complexity for all Java resources, contained in the virtualized access by the bag. To apply the function to all revisions, an analysis needs to traverse the entire revision sequence (using `add` and `remove`) and call `abstraction` after each step.

```
// Mutable data structure to maintain the input data.
val resources: mutable.Bag[(Path, Resource)] = ...

// The input of the program given in terms of add and remove.
def add(p:Path, r: Resource) = resources.add((p,r))
def remove(p:Path, r: Resource) = resources.remove((p,r))

// The output of the program (the function computing the abstraction).
def abstraction(): Int = {
    var result = 0
    for ((p, r) <- resources if p.endsWith(".java"))
        result = result + computeMCC(r)
    return result
}
```

Listing 3.3: A non-incremental solution

Such solution has scalability issues for obvious reasons. The actual function needs to be computed over and over again, for each traversed revision. Each call needs to exhaustively process the resources part of the particular revision.

Incremental Solution We can improve the solution by applying a simple program incrementalization, manually implementing the derivative (see Listing 3.4).

The solution operates on the input changes, given in terms of added and removed path-resources tuples. We use the change to update the resulting output abstraction, the total cyclomatic complexity, stored in an intermediate variable. In particular, the cyclomatic complexity of added Java resources is added to an intermediate variable `totalMCC` and the cyclomatic complexity of removed Java resources is subtracted.

```
    // Intermediate store for the abstraction.
    var totalMCC = 0

    def addResource(p: Path, r: Resource) =
        if(p.endsWith(".java")) totalMCC = totalMCC + computeMCC(r)

    def removeResource(p: Path, r: Resource) =
        if(p.endsWith(".java")) totalMCC = totalMCC - computeMCC(r)

    // The function computing the abstraction.
    def abstraction(): Int = totalMCC
```

Listing 3.4: An handmade incremental solution

The solution provides the same results as the non-incremental solution, but with different performance characteristics, better suited to compute the abstraction for each revision.

This program shows the essence of a basic program incrementalization, when implementing the derivative manually. It handles incoming changes efficiently, potentially producing outgoing changes. Outgoing changes may be connected to other

processing components (as incoming changes) in that more complex processing may be composed. The underlying idea is more general and can be expressed in terms of abstract algebra and certain properties of the applied functions.

3.3.2 General Incrementalization

The example problem that we have presented so far computes a pure function f on all revisions. The function takes a single revision and produces an abstraction of it. In our example, this function f takes the bag of resources (input) and produces an integer (output). The type signature of such function is `Bag[(Path, Resource)]` $\rightarrow$ `Int`.

The function f and the data types for input and output are generic parts of a generally applicable incrementalization framework. If the function and types can handle changes and act incrementally, performance benefits may kick in. This section will describe the limits of such generic parts, which we call the **core interface** for functions and types. We support this discussion by concrete instances, needed to compute the sum of the cyclomatic complexity, but the interface is not limited to cyclomatic complexity.

Our final prototype will follow an advanced strategy, building another layer on top of the core interface. We will implement the primitives of extended map-reduce (e.g., map, filter, group, join, count or sum) and repository resource access (git-diff), on top of the core interface. More complex processing, like computing and summing up the cyclomatic complexity, can be composed out of the primitives. We thereby do not need to rewrite any primitive incremental functionality from scratch. Such primitives will be discussed in the context of our prototype.

Representing Data and Change

The first part of the core interface covers the input and output data types for a function. The requirements are different from typical data types, as we also need a mechanism to capture data's change.

We define data and changes to follow the axioms of algebraic structures. An algebraic structure is defined over a set S and closed under a binary operator $\odot : S \times S \rightarrow S$ (Axiom 1). For each data type that we use, we need a corresponding algebraic structure.

Axiom 1 (Closed under $\odot$) *If $x, dx \in S$, then $x \odot dx \in S$.*

Data and changes are elements in S. If x is some data and dx some change, and both are contained in S, we consider $x \odot dx$ to be the data x after the change dx has been applied. According to Axiom 1, the result is included in S. Hence, we can represent arbitrary changes by a sequence of elements in S, fold by the $\odot$ operator.

We do not distinguish between data and changes (every data entry can also be understood as a change and vice-versa) because typically, there are algebraic structures applicable to represent both. There are also methods that distinguish between data and changes but without immediate benefits within the context of our work (see [CGRO14] for change structures that distinguish between data and changes).

Example Output To make this more concrete, we now transfer such insights to our running example. We start describing the output type of the function. We use an algebraic structure, defined on the set of integers, S, with + (summation) being the operator $\odot$. Consider the following example of changing data. We read from left to right[5]:

$$\overbrace{0}^{\text{data}}\ \overbrace{\odot 7 \odot -5}^{\text{change}}$$

The sequence represents data starting at 0, which is changing to 7 (by change 7), and to 2 (by change -5). Summarizing the previous discussion: For our running example, the output data and its changes can both be represented by using a plain integer as data type. This is a relevant insight to a technical solution.

We want to emphasize that this illustration is tailored to our running example. In our final map-reduce prototype, such limitations on types will hardly be noticed, since one typically uses collections that allow flexible types of contained elements. Algebraic structures are only relevant when understanding the limitations of the underlying core interface.

Example Input For our example's input data type, we use a bag of path-resource tuples (our most important collection type). We define the algebraic structure on the set of bags T. To enable representing changes in terms of added and removed elements by T, elements in the bag need to be tagged to indicate on removal. We write such elements as $\overline{r_2}$. The operator $\otimes$ (we use different symbols for this algebraic structure) is defined to be the bag union and cancels out corresponding adds and removes. For instance, $\{\overline{r_2}\} \otimes \{r_2\}$ is the same as an empty bag $\{\}$. Consider the following example of changing data:

$$\overbrace{\{\}}^{\text{data}}\ \overbrace{\otimes\{r_1, r_2\} \otimes \{\overline{r_2}, r_3\}}^{\text{changes}}$$

The sequence starts on an empty bag $\{\}$ and then adds r_1 and r_2. Hereafter, r_2 is removed and r_3 is added. The elements r_1, r_2 and r_3 stand for path-resources tuples in this example.

Algebraic structures are useful to formalize sequences of changes. At the same time, implementing an algebraic structure allows plugging custom data types into the core interface of our incrementalization prototype. We will cover more algebraic structures in the remainder of this thesis, but the most central structure for our map-reduce prototype will be the bag. Have a look at our hand made incremental solution again. We may now define both, function and derivative, having the type signature `Bag[(Path, Resource)]` $\rightarrow$ `Int`. This will later be a property of a self-maintained derivative.

Axioms of the Group Operator Four more axioms on the operator $\odot$ give rise to an algebraic structure. In the remainder of this work, we will focus on algebraic

[5]There are formal ways to define such sequences of changes. We go for a very intuitive notation, from left to right. In the following up discussion, transferring insights to the revision history, we will be more formal.

	Associativity	Identity	Invertibility	Commutativity
Monoid	•	•		
Com. Monoid	•	•		•
Group	•	•	•	
Abelian Group	•	•	•	•

Table 3.1: Axioms of Algebraic Structures

structures where all axioms hold. Such structures are called Abelian groups. However, for understanding different methods to incrementalization, this background section also discusses other structures. Axioms may limit the core interface and have relevant technical implications.

Axiom 2 (Associativity) *For a, b and c in S, $(a \odot b) \odot c = a \odot (b \odot c)$ holds.*

Axiom 3 (Identity) *For each a, there exists an identity element z for that $a \odot z = a$ and $z \odot a = a$ holds.*

Axiom 4 (Invertibility) *For each element a, there exists an inverse element a^{-1} in that $a \odot a^{-1} = z$.*

Axiom 5 (Commutativity) *For a and b in S, $a \odot b = b \odot a$.*

Depending on which of the axioms hold, algebraic structures are called: *Monoids* (ass. and id.), *groups* (ass., id. and inv.), *commutative Monoids* (ass., id. and com.) and *Abelian groups* (all). See Table 3.1 for a summary. We do not claim for completeness, as there are other algebraic structures that are not directly relevant to this discussion.

We will do a short examination of the important benefits and drawbacks related to the structures listed above.

Monoid and Commutative Monoid A monoid (or commutative monoid) is used to formalize monotonous growth of data and to express the corresponding change. The relevant characteristic is the missing inverse element, which is mandatory to undo previous changes (Axiom 4).

(Commutative) monoids are often used in the processing of (ordered) event streams (e.g., [BROL14]). Such streams are continuously growing, while undoing previous events is disregarded. For ordered event streams, monoids, S can be defined as the *set of lists* closed under *list concatenation*. For unordered event streams, commutative monoids, S is defined as the *set of sets* closed under the *set union*. Monoids are also used to formalize monotonous growth of the Datalog fix-point operator [AEJO19]. Datalog will be discussed in the next chapter of this thesis.

For typical abstractions of a repository, applied in MSR/ESE, the usage of commutative monoids may have some interesting use cases, possibly relaxing the limitations of our current prototype.

For instance, if we want to compute an abstraction of the past or future of a given revision, e.g., summing up the distinct authors in the past, we have monotonous behavior of the data when traversing the revision history. We may decode data and

change as commutative monoid, by the *set of sets* closed under the *set union*. The identity element will be the *empty set*. In particular, a function can be applied to sets that include all previous SHAs (commits), and derive all kinds of useful abstractions from it. On monoids, some applied functions will become more attractive, for instance, min or max aggregation will have efficient derivatives.

We did not yet examine implications of using (commutative) monoids, and also did not consider switching between algebraic structures with different axioms. However, we assume that this will be a good extension of the capabilities of our method.

Group and Abelian Group A group (or Abelian Group) is used to formalize data that allows undoing previous changes (including Axiom 4). If the input data of a function is defined in this manner, it allows flexibility in terms of growing and shrinking data, at the price of some more expensive computations. For instance, functions implementing min and max will be less efficient, compared to those working on (commutative) Monoids.

Groups and Abelian Groups are used to formalize general program and database incrementalization, where data structures (like collections and tables) are modified in terms of add and remove operations (e.g., [CGRO14, GGMS97]). Groups can be used to formalize lists, where the order in which the updates apply matters. Abelian Groups can be used in the formalization of bags where order does not matter.

Since this thesis primarily focuses on static abstractions, computed on the resources of single revisions, monotonously growing data is not possible. We need data types that reflect resource changes in terms of added and removed resources between revisions. Commutativity helps to align with the acyclic history. Limitations of working with Abelian groups will later get clear. Working with time windows shares the same growing and shrinking characteristics which calls for Abelian groups.

Processing Data and Change

To complete the core interface, we describe the generic aspects of the applied function f, specified as $f : S \to T$. Both, input and output type S and T need a corresponding algebraic structure to have a representation for data and changes, defined by the tuples $(S, \odot, z_S)$ and $(T, \otimes, z_T)$. The elements of the tuples are the set, the operator, and the identity element respectively. The inverse element is always written as x^{-1}.

Derivatives Incrementalization is based on the idea of having *derivatives* for functions (see Definition 1).

If we have $x \odot dx$, where $x \in S$ is some data, $dx \in S$ some change, we can optimize a function application $f(x \odot dx)$ by reusing the previous result $f(x)$ while the derivative $f'(x, dx)$ computes the output change when the input x is changed by dx. The derivative f' needs to assure that the semantics of the original f is preserved. Formally, this means $f(x \odot dx) = f(x) \otimes f'(x, dx)$ needs to hold.

Definition 1 (Derivative) *Having $x, dx \in S$ and a function $f : S \to T$, a function $f' : S \to T$ is a derivative if $f(x \odot dx) = f(x) \otimes f'(x, dx)$ holds.*

Definition 1 is the key insight to incrementalization. When replacing the left-hand-side of the equation with the right-hand-side, we may save resources if the computation of $f(x)$ can be reused. Such reusing can be done recursively on change sequences.

$$\overbrace{f(x \odot dx)}^{\text{non-incremental}} = \overbrace{\overbrace{f(x)}^{\text{previous}} \otimes \overbrace{f'(x, dx)}^{\text{derivative}}}^{\text{incremental}}$$

Many functions have a known derivative f' that enables this sort of computation.

Self-maintained Derivatives We see that x is a mandatory input to both functions f and f'. This is critical to solutions, e.g., limiting how data x is stored and distributed. For instance, optimizing $f(x)$ and $f'(x, dx)$ to run on different hardware is not possible without some data sharing mechanism, sending x.

However, there are derivatives that are self-maintained, i.e., the derivative is independent of x (Definition 2). In this case, we meet the definition of a homomorphism, another concept from abstract algebra. We call this function h and say that the function 'is' a self-maintained derivative.

Definition 2 (Self-maintained derivative) *Having $x, dx \in S$ and a function $h : S \to T$, the function h is a self-maintained derivative if $h(x \odot dx) = h(x) \otimes h(dx)$ holds.*

In practice, this implies that input data x does not need to be maintained while processing a sequence of changes by h.

$$\overbrace{h(x \odot dx)}^{\text{non-incremental}} = \overbrace{h(x) \otimes h(dx)}^{\text{self-maintained, incremental}}$$

Many functions will conform to such more efficient formalization. Take our running example of the computation of the cyclomatic complexity as an instance, where only the added and removed path-resources tuples (dx) are needed, while maintaining the bag of overall path-resources tuples (x) is not needed. Function h filters for java resources and sums up the cyclomatic complexity of changed resources, where added resources are positive numbers and removed resources are negative numbers.

While this is just an example of a concrete self-maintained derivative, we will later provide a set of primitives that have self-maintained derivatives to allow a composition of complex processing. In the case of our running example, we may use self-maintained derivates of map-reduce functionality `map`, `filter` and `sum` to compose the abstraction of the revision. We do not need to write the derivatives manually.

Operations Conforming the Core Interface

In the remainder of this thesis, we will focus on incrementalizing primitives of extended map-reduce and on git-diff. The primitive functions are sufficient to compose complex processing, and they can be plugged into our core interface. We will cover

further technical implications for the prototypical realization in Sec. 3.4.3. The evaluation in Sec. 3.5 measures and compares performance benefits of different methods for the running example on the cyclomatic complexity.

3.4 Technical and Methodological Improvements

The main contributions of this work are structured as follows. In Sec. 3.4.1, we use the background on abstract algebra to associate data with revisions, and changes with parent relations. In Sec. 3.4.2, we use the background on derivatives and self-maintained derivatives for the processing of changes instead of the corresponding data. In Sec. 3.4.3, we describe the challenges of implementing a working prototype that is embedded into Scala. The prototype uses the previous ideas on incrementalization. We use the prototype in the evaluation and show that this design improves the scalability.

3.4.1 Representing Repository History

Opposed to regular program incrementalization practice, which focuses on a sequence of changes, we need to consider the acyclic nature of the repository history.

We consider the following aspects (Sec. 3.4.1, 3.4.2 and 3.4.3) as design decisions. The design suffices to implement typical computations of MSR/ESE using our final prototype. Later evaluation will show that the design improves scalability.

Graph Structure The repository history consists of a set of revisions and a set of parent relations. This is a graph $G = (E, N)$. Revisions are points in the history N. Parent relations connect one point to another and represent an evolution E. Parent relations are created by actions, such as committing, forking and merging. We prefer the term *revision* instead of *commit* to better distinguish nodes from edges in the graph. Since it is impossible for a revision to be a parent of itself, we have an acyclic graph.

Data and Changes To transfer the insights on incrementalization to repository history, we associate data with revisions and changes with parent relations. This novel association is given in terms of the total functions for nodes $\Pi : N \to S$ and edges $\Pi' : E \to S$. The tuple of the associations makes up our data structure.

For describing data and changes, we focus on Abelian groups $(S, \odot, z_S)$. The usage of Abelian groups is motivated by our typical showcase of functions that operate on the bag of resources contained in a revision, which may grow or shrink over the revision history. We go for Algebraic groups because it provides a proper inverse (shrinking). Commutativity helps to align with the acyclic history[6].

Properties of Π and Π' We initialize the associations according to the following two ideas on how data Π and its change Π' manifest in repositories.

- We initialize the association Π so that it stores data (maintains the actual 'payload' of the data structure) for each revision. It is a generic part of the

[6]Other than that, if we aim at analyzing all preceding revisions of a given revision, we may go for a commutative monoid, since such data grows monotonously. We did not yet examine the second use case in this thesis. All structures may come with other limitations and benefits, we focus on static abstractions on single revision, and Abelian groups.

data structure. In our running example, we instantiate it as the bag of path-resources tuples, or the cyclomatic complexity metrics.

- We initialize the association Π' (the change) according to Π and G. The formalization of data and changes, given by an Abelian group, can be used to guide the definition of change on an edge in E given by Π'. See Definition 3.

Definition 3 (Edges) *Let N, E, Π, S, $\odot$ be the repository history and Abelian group, for every edge $(n_1, n_2) \in E$, the change association is defined by $\Pi'((n_1, n_2)) = \Pi(n_2) \odot \Pi(n_1)^{-1}$.*

This definition says that for getting the change between two nodes, we form the inverse of the first node, and combine it with the second node. It is a conventional difference described in terms of abstract algebra. From Definition 3 it logically follows that when traversing a path P through the graph, connecting n_1 and n_2, the accumulated changes $\Sigma^{\odot}_{e \in P} \Pi'(e)$ will be consistent with the overall change between the connected revisions $\Pi(n_1)^{-1} \odot \Pi(n_2)$. This leads to Lemma 1.

Lemma 1 (Paths) *Let N, E, Π, Π', S, $\odot$ be the repository history with Abelian group, for every directed path P between two nodes $n_1, n_2 \in N$, $\Pi(n_2) = \Pi(n_1) \odot \Sigma^{\odot}_{e \in P} \Pi'(e)$ holds.*

Lemma 1 offers a way to circumvent maintaining (or computing) the entire associations Π and Π' which are central to our data structure. Using Definition 3, we can turn Π into Π'. Using Lemma 1 we can turn Π' back into Π if needed[7]. We thereby have the option to only use one of both associations when working with the data structure. It may often be the case that the changes Π' will be easier to maintain (and compute if we can use derivatives). Lemma 1 will be the theoretic background for a *Data Traversal* (introduced later in the more technical sections).

Visual Example Figure 3.1 shows the conceptual idea of associating data and changes with the graph visually.

The circles correspond to the revisions $N = \{A, B, C\}$ and the directed edges correspond to the parent relations $E = \{(A, B), (B, C), (A, C)\}$. The lower-case letter correspond to data $u, v, w \in S$. In Figure 3.1, data can be found inside the circles. Changes are annotated to the edges of the graph and defined according to Definition 3.

3.4.2 Processing Repository History

We now transfer the capabilities of more efficient processing mechanisms using (self-maintained) derivatives, to a data structure defined according to the previous section, in terms of Π and Π'. We now distinguish between input S and output T for an association respective to an applied function $f : S \to T$.

[7]In particular, if we have $\Pi(n_1)$ for a single node n_1, we can compute Π on every node reachable from n_1 by just using the changes Π' on necessary edges. The single node n_1 is later referred to as a checkpoint.

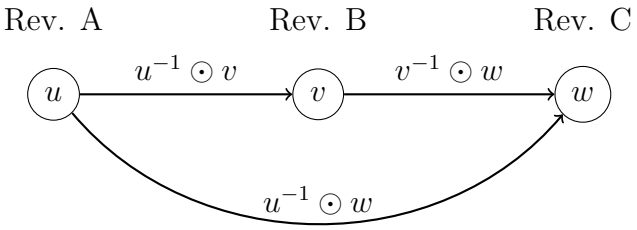


Figure 3.1: Associating data and changes with the revisions A, B and C and the parent relations (A,B), (B,C) and (A,C): The data associated with nodes is $\Pi(A) = u$, $\Pi(B) = v$ and $\Pi(C) = w$. The change associated with edges is $\Pi'((A,B)) = u^{-1} \odot v$, $\Pi'((B,C)) = v^{-1} \odot w$ and $\Pi'((A,C)) = u^{-1} \odot w$, defined according to Definition 3.

The associations of an input data structure Π_S and Π'_S are assumed to be correctly defined according to Definition 3. Our aim is to apply a function f in that an output association is $\Pi_T(n) = f(\Pi_S(n))$ for all $n \in N$. This is the way we assume a data structure to work. By chaining such function calls, we can compose the computation of static abstractions of single revisions. We apply the function to all $n \in N$.

Definition 4 (Function Application) *Let N, E, Π_T, Π_S, S, T be the repository history and Abelian groups, for input and output associations, for every node $n \in N$, the data association after an application of f is $\Pi_T(n) = f(\Pi_S(n))$.*

This computation may be inefficient. We prefer computing the new change $\Pi'_T(n_1, n_2)$ as a derivative, like $f'(\Pi_S(n_1), \Pi'_S(n_1, n_2))$, instead. If this is correct, we can use Lemma 1 to recover Π_T from the efficiently computed Π'_T. Correctness of the more efficient computation follows from our previous definitions:

$$
\begin{aligned}
\Pi'_T(n_1, n_2) &\stackrel{?}{=} f'(\Pi_S(n_1), \Pi'_S(n_1, n_2)) && \\
\Pi_T(n_1) \otimes \Pi_T(n_2)^{-1} &\stackrel{?}{=} f'(\Pi_S(n_1), \Pi'_S(n_1, n_2)) && \text{(ap. Definition 3)} \\
f(\Pi_S(n_2)) \otimes f(\Pi_S(n_1))^{-1} &\stackrel{?}{=} f'(\Pi_S(n_1), \Pi'_S(n_1, n_2)) && \text{(ap. Definition 4)} \\
f(\Pi_S(n_2)) &\stackrel{?}{=} f(\Pi_S(n_1)) \otimes f'(\Pi_S(n_1), \Pi'_S(n_1, n_2)) && \text{(ap. } \otimes f(\Pi_S(n_1))) \\
f(\Pi_S(n_1) \odot \Pi'_S(n_1, n_2)) &\stackrel{?}{=} f(\Pi_S(n_1)) \otimes f'(\Pi_S(n_1), \Pi'_S(n_1, n_2)) && \text{(ap. Lemma 1)} \\
f(x \odot dx) &\stackrel{?}{=} f(x) \otimes f'(x, dx) && \text{(renaming)} \\
f(x \odot dx) &= f(x \odot dx) && \text{(ap. Definition 1)}
\end{aligned}
$$

This results in Lemma 2. Lemma 3 follows accordingly.

Lemma 2 *If a function f has a derivative f', for every edge $(n_1, n_2) \in E$ in the graph, $\Pi'_T(n_1, n_2) = f'(\Pi_S(n_1), \Pi'_S(n_1, n_2))$ holds.*

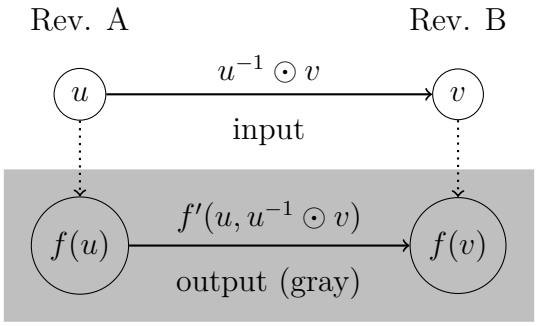


Figure 3.2: Application of a derivative with input and output graph

Lemma 3 *If a function h is a self-maintained derivative, for every edge $(n_1, n_2) \in E$ in the graph $\Pi'_T(n_1, n_2) = h(\Pi'_S(n_1, n_2))$ holds.*

These two lemmas provide the essence of optimized application of a function to all revisions of a repository, being a technical alternative to a computation following Definition 4.

Instead of computing Π_T for all $n \in N$, we can go the indirection over Π'_T, efficiently computed by the derivatives, and then recompute Π_T according to Lemma 1 along the parent relations.

The limitation of regular derivatives, that need Π at the time of applying f', can be avoided by recovered it on the fly according to Lemma 1. This still circumvents applying f to every single revision.

Visual Example We see a visual example of the exchangeability, for a regular derivative, in Figure 3.2. Data associations are shown insider the nodes. The change associations are shown as edge labels. The function application is shown as dotted arrows from top to bottom (input to output).

3.4.3 Implementing the Topleet Prototype

The previous sections illustrate how to formalize general program incrementalization and how to transfer these insights to revisions and parent relations of a repository's history. In this section, we show how such insights can be turned into a concrete data structure for processing repository history, as a prove-of-concept, and for running an evaluation. The prototype that we show implements concrete derivatives of map-reduce, that can immediately be used.

The following code is integrated into standard Scala. It shows our Topleet prototype computing the cyclomatic complexity on all Java file revisions in the repository libgdx/libgdx.

```
1 // Git initialization.
2 val shas: Leet[SHA] = git("libgdx/libgdx")
3
4 // Accessing the resources of each commit.
4 val resources: Leet[Bag[(Path, Resource)]] = shas.resources()
```

```
5
6 // Filtering for Java.
7 val javas: Leet[Bag[(Path, Resource)]] = resources
8   .filter { case (path, resource) => path.endsWith(".java") }
9
10 // Computing the metrics.
11 val mcCabe: Leet[Bag[Int]] = javas
12   .map { case (path, resource) => computeMCC(resource)}
11
12 // Summing up the metrics.
13 val summed: Leet[Int] = mcCabe.sum()
```

Listing 3.5: Topleet Cyclomatic Complexity Solution: The part from line 4 to 10 corresponds to the map-reduce steps of the DJ-Rex solution, shown in Listing 3.1.

The code uses the Topleet data structure `Leet[S]`, which maintains the repository history as a graph in the background. Revisions are nodes and parent relationships are edges. The data structure associates each node of this background graph with one data entry of the generic type `S` for which we have a corresponding Abelian group. The Abelian group is inferred during compile time using a Scala mechanisms called implicit parameters.

Method calls on the data structure invoke processing steps that return new data structures with altered data and change associations. This enables a fluent API syntax that sequentially applies processing steps to the data entries of all revisions, similar to map-reduce. Such processing steps are backed by a core interface for applying regular and self-maintained derivatives.

For clarity, we assign the intermediate results in Listing 3.5 to placeholders (`val`) annotated by the type. The solution starts with a call to `git`, initializing the first data structure with the revision history of repository libgdx/libgdx used as background graph. It associates each node of the background graph (i.e., a revision) with a data entry of type `SHA` reflecting the revision's identity. Method `resources()` is invoked on this data structure to read out the available path-resource tuples for each `SHA`, i.e., the revision's corresponding resources at this point in the history. In the background, this call relies on the self-maintained derivative *git-diff*. The returned data structure maintains data entries of type `Bag[(Path, Resource)]`. The type `Bag[E]` is a bag of elements of type `E`. `Path` is a path to a resource and `Resource` is a pointer to repository content providing an input stream. The next steps invoke `map`, `filter` and `sum` according to the standard semantics of bags; `computeMCC` is a foreign function of type `Resource` $\rightarrow$ `Int` to compute the cyclomatic complexity on a single resource; it is the parameter of the high-order `map` function.

Topleet processes the libgdx/libgdx history in around 3.2 minutes (DJ-Rex Incremental takes 6.5 minutes, LISA 12.0 and DJ-Rex 449 minutes). Topleet includes the change processing (incrementalization), distribution and reduction of redundancies under the hood, but looks almost similar to the basic map-reduce solution, such as DJ-Rex.

Core Interface

Topleet builds up extended map-reduce on top of a simple core interface. The core interface consists of operations for:

- applying derivatives (`tmap`),
- applying self-maintained derivatives (`tmapHom`),
- merging two data structures with the same background graph according to the operator of the Abelian group (`merge`) and
- initializing a data structure with arbitrary data.

Background Graph

The Topleet data structure `Leet[S]` hides the repository history under the hood in terms of the *background graph*. The background graph is created during the initialization of the data structure and remains unchanged from this point on. It consists of i) nodes `N` that represent individual revisions in the history, and of ii) directed edges `(N, N)` connecting very similar revisions by the parent relations.

Conceptually, the background graph is not limited to repository history and may also reflect other evolving artifacts, for instance, the versions of a JAR file and the respective semantic versioning relation. Versioning practice, like the usage of branches, may affect the topology of the background graph being a sequence, tree or acyclic graph.

Change Collection

According to our discussion on the associations, Π and Π', our data structure prefers to work with Π' instead of Π whenever possible. Instead of using data of type `S`, at every node `N`, represented by, e.g., `Map[N, S]`, Topleet prefers a collection of data changes represented as `Map[(N, N), S]`. We refer to this as *change collection*.

An Abelian group, corresponding to type `S` of `Leet[S]`, is used to initialize the underlying elements of the change collection according to Definition 3. For each background graph edge `(`n_1`,` n_2`)` and the corresponding data association $\Pi(n_1)$ and $\Pi(n_2)$, a change element $\Pi'(n_1, n_2) = \Pi(n_2) \odot \Pi(n_1)^{-1}$ is created during the initialization of the data structure.

Implementing Abelian Groups

To derive the change collection for, let's say, `Leet[Int]`, we need an Abelian group for `Int`. While our prototype typically works with bags as data type, we also show Int for illustration here.

The Abelian group for natural numbers `Int` defines addition to be the operator $\odot$, negation to be the inverse and 0 to be the zero element z. An implementation can be found in Listing 3.6.

```
1 case class MyInteger extends AbelianGroup[Int]{
2
3   override def zero(): Int = 0
```

```
4
5     override def op(a: Int, b: Int): Int = a + b
6
7     override def inv(a: Int): Int = −a
8
9 }
```

Listing 3.6: Implementing an Abelian group for type `Int`

This Abelian group is passed into a data structure automatically whenever type `Int` is used to initialize it, or if `Int` is the output of an applied derivative. This is done by an implicit parameter. Without such mechanism, using the data structure is more cumbersome because corresponding Abelian groups have to passed explicitly whenever needed.

Composing Abelian Groups

For covering all types of our showcase in terms of Abelian Groups, such as `SHA`, `Bag[(Path, Resource)]` and `Bag[Int]`, we rely on a mechanism for composition.

One of the fundamental types that we use for such composition is `Map[K, V]` which has a corresponding Abelian group if there is a 'nested' Abelian group for `V`. The Abelian group for `Map[K, V]` is defined as follows: The operator e^{-1} inverts the values of the map by the nested e_V^{-1} operator; the $\odot$ operator merges two maps in that it combines the values with the same key by the nested $\odot_V$ operator, filtering out nested z_V elements; operator z returns an empty map.

Accordingly, we can use the Abelian group for `Int` to compose an Abelian group for `Map[K, Int]`. The type `Map[K, Int]` can also be considered as an alias for `Bag[K]` assigning bag element `K` to the number of its occurrence, which may also be negative. This is effectively the same as tagging removed elements, which we have used in Sec. 3.3 to distinguish between added and removed elements in bags. We use this Abelian group for the data structure `Leet[Bag[(Path, Resource)]]` and `Leet[Bag[Int]]` in our cyclomatic complexity solution. The final user of our prototype will typically work with bags.

Other than that, we have a corresponding Abelian group for tuples `(V1,V2)` that can be composed out of an Abelian group for `V1` and `V2` by delegating to the nested Abelian groups.

Escaping Abelian Groups

The processing of a Git repository, as shown in the solution, usually starts with the initial data structure `Leet[SHA]`. However, the type `SHA` misses a corresponding Abelian group; hence, we miss a way to represent it as change collection. In this case, we fall back to non-incremental processing. Effectively, we wrap type `SHA` in a single element `Bag[SHA]`, having a Abelian group, to ease the implementation of our prototype.

However, finding a reasonable decomposition of `S` into parts that change little along the edges of the background graph is the premise to efficient processing changes. While the repository content is predestined to be decomposed into bags of resources that change little (i.e., by `Bag[(Path, Resource)]`), decomposing type

`SHA` is not beneficial. Neither would decomposition imply small changes, as succeeding SHAs are not trivially related, nor would there be derivatives with useful properties.

Data Traversal

Our data structure processes changes, associated with parent relations, instead of data, associated with revisions. Hence, data associated with revisions needs to be restored whenever needed. This may be required when reading out the content of the data structure at revisions or when applying a regular derivative $f'(x, dx)$ that requires x.

Reading out data of Π happens according to Lemma 1. Changes Π' can be computed and fold along a path in the background graph, in that data Π, associated with the revision and reachable over some known data association, can be recomputed. We refer to this as *data traversal*. The traversed data can be listened. Immutable types help to avoid visiting nodes twice or copying the data when the background graph branches.

Checkpoint Collection

A problem of a data traversal, present in Lemma 1, is that it needs at least one data association to start with. Without such reference point, the change collection cannot be turned into data again. To enroll a data traversal, the Topleet structure maintains a second *checkpoint collection* processing the data association of one node in every connected component of the background graph. A type like `Map[N, V]` can be used for this. The checkpoint collection is processed in analogy to the change collection.

Applying Derivatives

For the processing, the way in favor is to work on the change collection by applying (self-maintained) derivatives. The data structure enables this by the core operation `tmap` and `tmapHom`. The prefix 't' stands for topological. We use this naming to disambiguate between default map calls on the type `Bag[V]` of `Leet[Bag[V]]`, one of map-reduce's primitive function.

If having a regular derivative, the interface `tmap` of `Leet[S]` enables applying a function $f' : (S, S) \to T$ to produce `Leet[T]`. The two parameters of the applied function are x and dx, i.e., the data and its change on an edge. The function needs to conform Definition 1.

If having a self-maintained derivative, the interface `tmapHom` of `Leet[S]` enables applying a function $h : S \to T$ to produce `Leet[T]`. The parameter is dx, i.e., the change on an edge. We need to be sure that the function conforms Definition 2.

Both calls to the core interface turn the change collection into the new change collection according to Lemma 2 and Lemma 3. The checkpoint collection is modified accordingly. The background graph is preserved during such invocation.

Accessing Resources (Git-Diff)

Most functions that compute static abstractions of a single revision start on the bag of path-resource pairs contained in the revision. We would expect some functionality f giving us the resource of a given revision (in our prototype the function is called `resources`). We prefer to rely on bags, for which we have an Abelian group. We define the input of f to be of (the escaped) type `Bag[SHA]` and the output to be of type `Bag[(Path, Resource)]`.

Since we favor derivatives, we need f'. The function *git-diff* exactly fulfills the requirements of a self-maintained derivative f'. We can use `tmapHom` to apply it.

3.4.4 Map-Reduce Operations

A second layer on top of the core interface implements the primitive functions of map-reduce, all realizable as (self-maintained) derivatives. Such functions can be used to compose more complex processing. From this point on, end-users will not get in touch with incrementalization at the core interfaces, but compose arbitrary processing using the primitives.

```
filter :  (K → Boolean) → Leet[Map[K, V]] → Leet[Map[K , V]]
map :  (K1 → K2) → Leet[Map[K1, V]] → Leet[Map[K2 , V]]
sum :  Leet[Bag[Int]] → Leet[Int]
count :  Leet[Bag[K]] → Leet[Int]
join :  Leet[Map[K,V1]] → Leet[Map[K,V2]] → Leet[Map[K,(V1,V2)]]
```

While the list shows regular aggregation, our prototype also allows applying group wise aggregation. Principles for the incrementalization are the same.

Example Cyclomatic Complexity

We now revise the computation steps of the cyclomatic complexity, working only on the change collection. A detailed picture of such processing is given in Figure 3.3.

The processing starts with a call to `git` taking the repository address as parameter. It initializes the data structure `Leet[SHA]` by creating the background graph and populates the corresponding change collection. The Abelian group used corresponds to type `Bag[SHA]` which is an alias for `Map[SHA, Int]` and the internal substitute for the escaped type `SHA`.

We exemplify such initialization on a simple background graph that consist of three succeeding commits A, B and C. We use letters to identify commits instead of real SHAs. To derive the first change collection, we first define the association of the background graph nodes: $\Pi(A) =$ `{(A→1)}`, $\Pi(B) =$ `{(B→1)}` and $\Pi(C) =$ `{(C→1)}`. Occasionally, we use a shorthand notation such bags, i.e., `{A}`, `{B}` and `{C}`. The first change collection lists the changes: $\Pi'(A, B) =$ `{(A→-1),(B→1)}` removing `A` and adding `B` (short $\{\overline{\texttt{A}},\texttt{B}\}$) and $\Pi'(B, C) =$`{(B→-1),(C→1)}` removing `B` and adding `C` (short $\{\overline{\texttt{B}},\texttt{C}\}$). From now on, the processing commences on the change collection with the elements `((A, B),` $\{\overline{\texttt{A}},\texttt{B}\}$`)` and `((B, C),` $\{\overline{\texttt{B}},\texttt{C}\}$`)`.

For the remaining steps of computing the cyclomatic complexity, we rely on `map`, `filter` and `sum`, which are map-reduce primitives that are self-maintained derivatives and thereby can immediately be applied to the change collection.

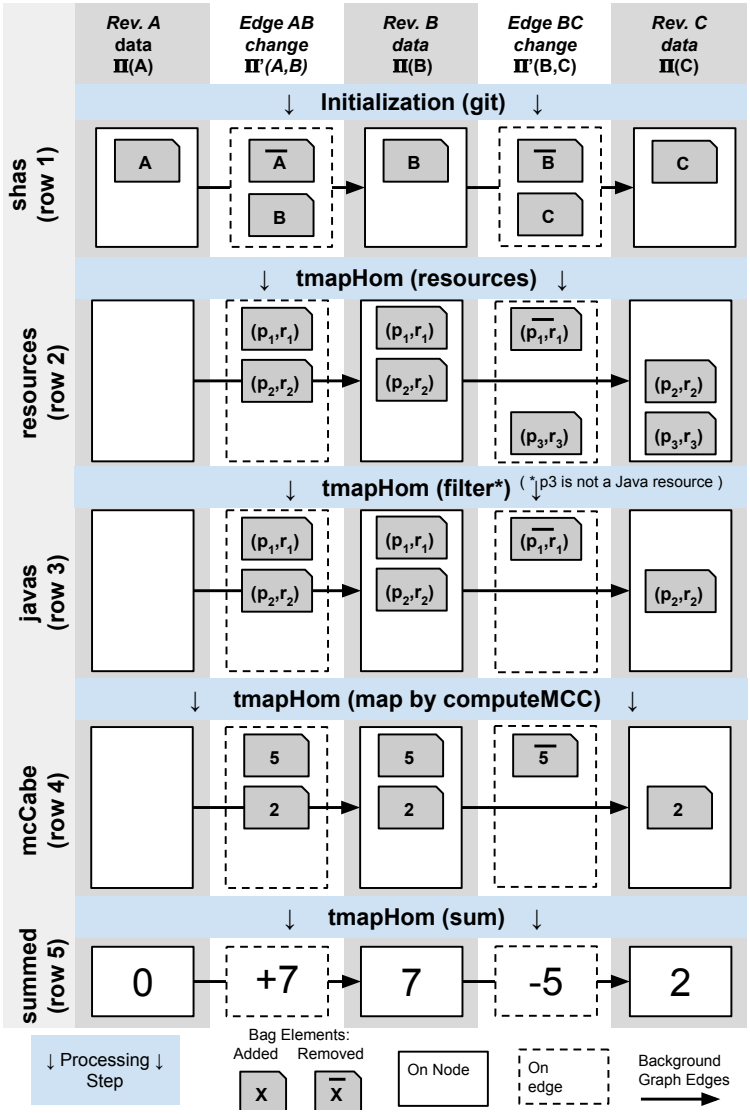


Figure 3.3: **Interchangeable processing of data and changes:** The plot depicts the processing steps (top to bottom) computing the cyclomatic complexity on three succeeding commits (left to right). **Row 1**: The data and changes are initialized according to an Abelian group for `Bag[SHA]`. **Row 1** $\rightarrow$ **2**: The path-resource tuples are extracted. Commit A is empty, commit B adds the tuples $(\mathtt{p}_1,\mathtt{r}_1)$ and $(\mathtt{p}_2,\mathtt{r}_2)$; commit C removes tuple $(\mathtt{p}_1,\mathtt{r}_1)$ and adds $(\mathtt{p}_3,\mathtt{r}_3)$. **Row 2** $\rightarrow$ **3**: We assume that path $\mathtt{p}_3$ points to a Bitmap; hence, the tuple $(\mathtt{p}_3,\mathtt{r}_3)$ is filtered out. **Row 3** $\rightarrow$ **4**: The metrics are computed given by type `Bag[Int]`. Resource $\mathtt{r}_1$ has a cyclomatic complexity of `5` and $\mathtt{r}_2$ a cyclomatic complexity of `2`. **Row 4** $\rightarrow$ **5**: Finally, the metrics are summed up. We apply a function between bags of natural numbers `Bag[Int]` and natural numbers `Int`. The change at edge AB is $\Pi'(A,B) = 7$ and the change at edge BC is $\Pi'(B,C) = -5$. All functions in this example have self-maintained derivatives (see the usage of `tmapHom`).

3.4.5 Advanced Infrastructure

Topleet can be realized on local or distributed computation infrastructures that provide map-reduce functionality on collections. In this section, we discuss how to adapt the core operations to efficient processing infrastructures. It is based on our experience implementing Topleet on Scala Collections and Apache Spark's Resilient Distributed Datasets (RDDs).

Lineage

Lineage refers to functions being chained and invoked on collections without materializing intermediate results [BF05, CCT09]. To guarantee lineage by an element-wise processing of the change and checkpoint collections, we adapt the original collection definition, e.g., `Map[(N, N), V]`, to plain sequences `Seq[((N, N), V)]`. It allows that changes on edges split into multiple collection elements. For instance, `((A, B),` $u * v$`)` can be split into the collection elements `((A, B),` u`)` and `((A, B),` v`)`.

Keeping u and v apart, or merging both, makes no semantic difference for the application of a self-maintained derivative. Combining two data structures works the same way.

However, merging corresponding changes and checkpoints in collections may decrease the size of the data and change (e.g., reducing $5 + 2 + 3$ to 10). Deciding if such merges should be applied, which necessary beaks lineage, depends on the use case. Breaking lineage may also decrease performance. We set corresponding defaults for the map-reduce functions and defer additional optimization to future work.

Distribution

We delegate the distribution to Spark's Resilient Distributed Datasets (RDDs) [ZCD+12]. To enable a partitioning by key, we set `S` in `Leet[S]` to `Map[K, V]` by default. This allows to use bags as the primary data type. The corresponding RDDs maintain key-value pairs for changes `RDD[((N, N), (K, V))]` and checkpoints `RDD[(N, (K, V))]`. We revise the interface for `tmapHom` and `tmap` accordingly.

Implementation of the core operations `tmapHom`, and `merge` on RDDs is straightforward, as no shuffling of data is required. For `tmap`, we face problems, since we need to enroll a data traversal. We apply a two-dimensional strategy to shuffle collection elements.

- We partition elements by hash of the element's key `K`.
- We partition elements by connected components of the background graph.

To apply `tmap`, the change and checkpoint collection elements are shuffled, in that those with the same key and those in the same connected component are moved to the same partition. Multiple assignments of `K` to `V` can be merged during the shuffle step according to the nested Abelian group for `V`.

Afterwards, `tmap` can be applied within a partition. It contains all necessary changes and checkpoints for a data traversal of the background graph.

Memoization

Applying a cost-intensive function (such as parsing) on the same input twice, can be circumvented by caching a function's input and corresponding output, or by applying the function on an inverted representation of the collections (e.g., applying `f` to the keys of `RDD[(K, Set[(V, N, N)])]`). Memoization is another open parameter that depends on the applied function and data. We reduce additional redundancies by such optimization.

3.5 Evaluation

We evaluate Topleet in computing McCabe's cyclomatic complexity for all Java file revisions in a repository. This task fits all related methods that we compare, and it is used for the presentation of LISA.

3.5.1 Solutions

We compare Topleet in different configurations with two manual migrations to distributed map-reduce, i.e., DJ-Rex and DJ-Rex Incremental (presented in Sec. 3.2 and available online); and with LISA (the solution code is presented in the publications [APG17, APPG19]). We cannot compare to Boa (presented in [DNRN15, DNRN13, NDNR14]) because we cannot reproduce the proprietary infrastructure behind it.

3.5.2 Software, Hardware and Default Parameters

All solutions are configured to the best of our knowledge. We follow a list of principles to assure that the evaluation is as objective as possible:

1) We use the same Java parser. 2) For DJ-Rex (Incremental), and Topleet we used Apache Spark for distribution. If running Spark on a single machine, we fully employ the capabilities by using the local mode with 16 cores. We use Kryo serialization with a buffer size of $512m$. 3) For Topleet and DJ-Rex (Incremental), we use one partition for each 100 revisions, with a minimum of 32 partitions to guarantee parallelism. 4) We patched the `tick-duration` of Apache Akka to 10 milliseconds to run LISA on Windows. 5) Solutions that depend on `computeMCC` use the same implementation. 6) We exclude the summation of the cyclomatic complexity to align the output granularity of all methods on the file level. 7) The output is fully persisted to a single storage system. We used the `collect` mechanism of Apache Spark and LISA's default CSV persistence. 8) The output of the methods differs: LISA persists metric values for linear ranges of the *flattened commit history*[8], DJ-Rex (Incremental) persists metric changes for the flattened commit history and Topleet changes for the commits of the acyclic commit history. 9) Depending on the distribution mode, all solutions are executed on the same hardware. The local evaluation is executed on an Intel Core i5-6600 @ 3.30GHz with 32GB memory, 64-bit, Windows 10. We isolate each run in a separate JVM. Distribution is evaluated using 4 or 7 Amazon EMR m5.xlarge on demand instance with 4 virtual cores and 16GB memory each. 10) Local measurements exclude the time for downloading a repository.

3.5.3 Variability

For Topleet, we explore the following configurations:

[8]Commits included in branches can be flattened into one linear sequence.

- **Topleet**: We refer to the solution performing best as Topleet. It uses Apache Spark, `tmapHom` for the `map` including memoization and `filter` preserving lineage.
- **Topleet (Mem. Off)**: Topleet without memoization.
- **Topleet Scala Collections (Mem. Off)**: The Scala Collections implementation, no memoization.
- **Topleet Data Traversal (Mem. Off)**: Topleet without memoization and `map` using `tmap` to enforce a data traversal on each application.
- **Topleet 4x m5.xlarge**: Topleet on one master and 3 cores.
- **Topleet 7x m5.xlarge**: Topleet on one master and 6 cores.

3.5.4 Subject Repositories

We execute the evaluation on a sample of 98 repositories based on the GHTorrent data set [GS12] from 2019-06-01[9] which is a mirror of the data exposed by the GitHub API. We sampled for Java project with more the 1000 watches and more than 10 developers according to the specification of the GHTorrent dump. We exclude the repository bytedeco/javacpp-presets as an outlier causing LISA to run into page faults. We excluded one repository requiring credentials. DJ-Rex hits the size limit for serialized results on repository SonarSource/sonarqube and wildfly/wildfly (29.000 and 28.000 commits), both repositories remaining in our sample, as we are interested in the performance of the other methods.

3.5.5 Correctness

The correctness of a Topleet variant can be checked during any processing step by comparing results to a very basic reference implementation, i.e., a local and non-incremental realization of the core operations (differential testing). Such implementation is trivial to write. We check such correspondence for different tasks on the repository sample.

3.5.6 Time

Since LISA is not distributed, we compare all methods when running on a single machine but still with all benefits of Apache Spark enabled in local mode. This also assures that we have a fair comparison of resource usage. The averaged time needed for all 98 repositories by a solution is shown in Figure 3.4. The averaged time needed for repositories grouped into exponentially growing commit count buckets is depicted in Figure 3.6. DJ-Rex misses time measurements for two repositories.

For repositories with low commit counts, all solution depicted in Figure 3.6 need a comparable amount of time. The time increases differently with increasing commit count. The methods are ranked as follows: DJ-Rex takes the most time (on

[9] `http://ghtorrent.org/downloads.html`

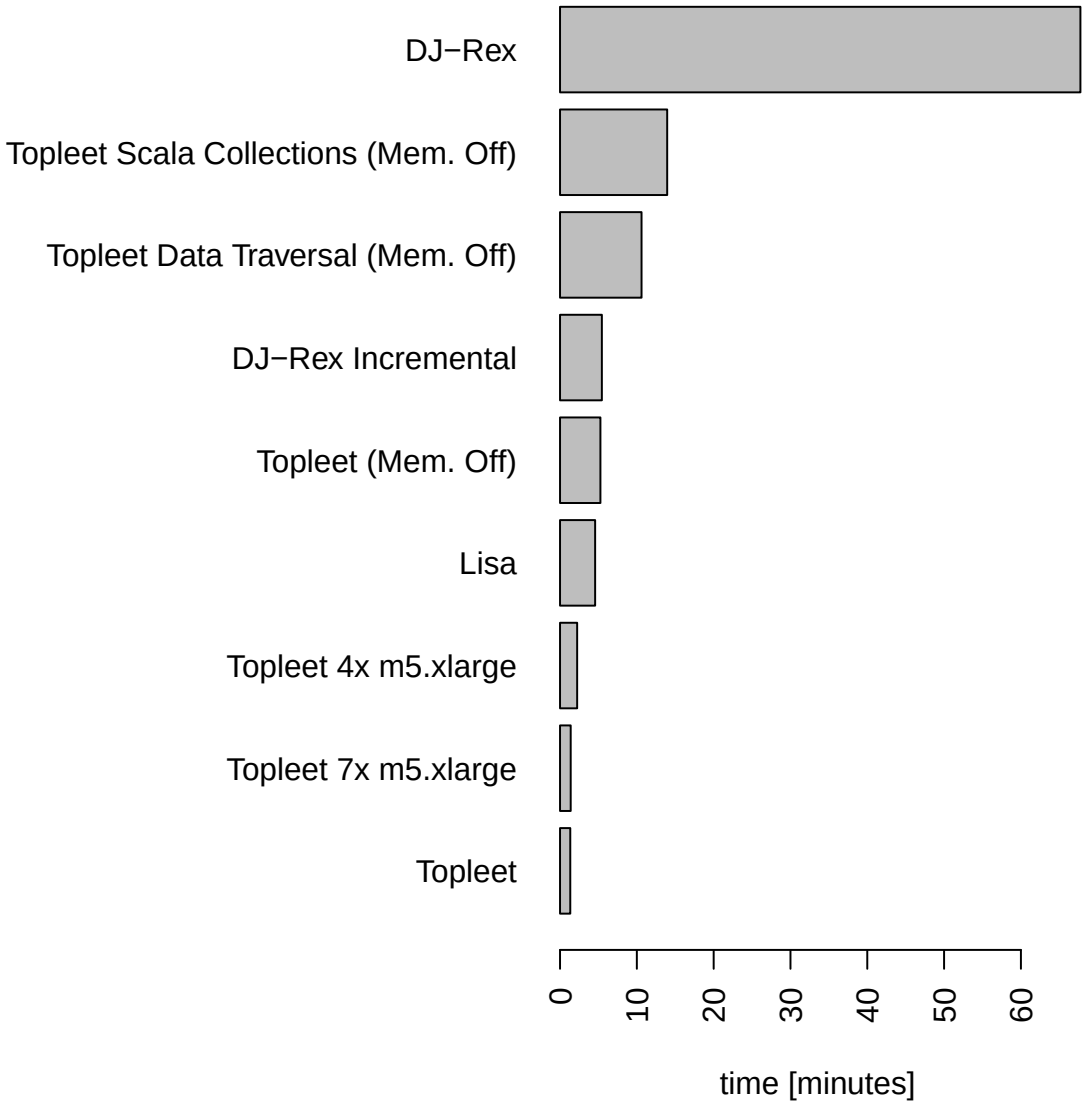


Figure 3.4: The average time of a method, running on one of the 98 repositories, sorted by time.

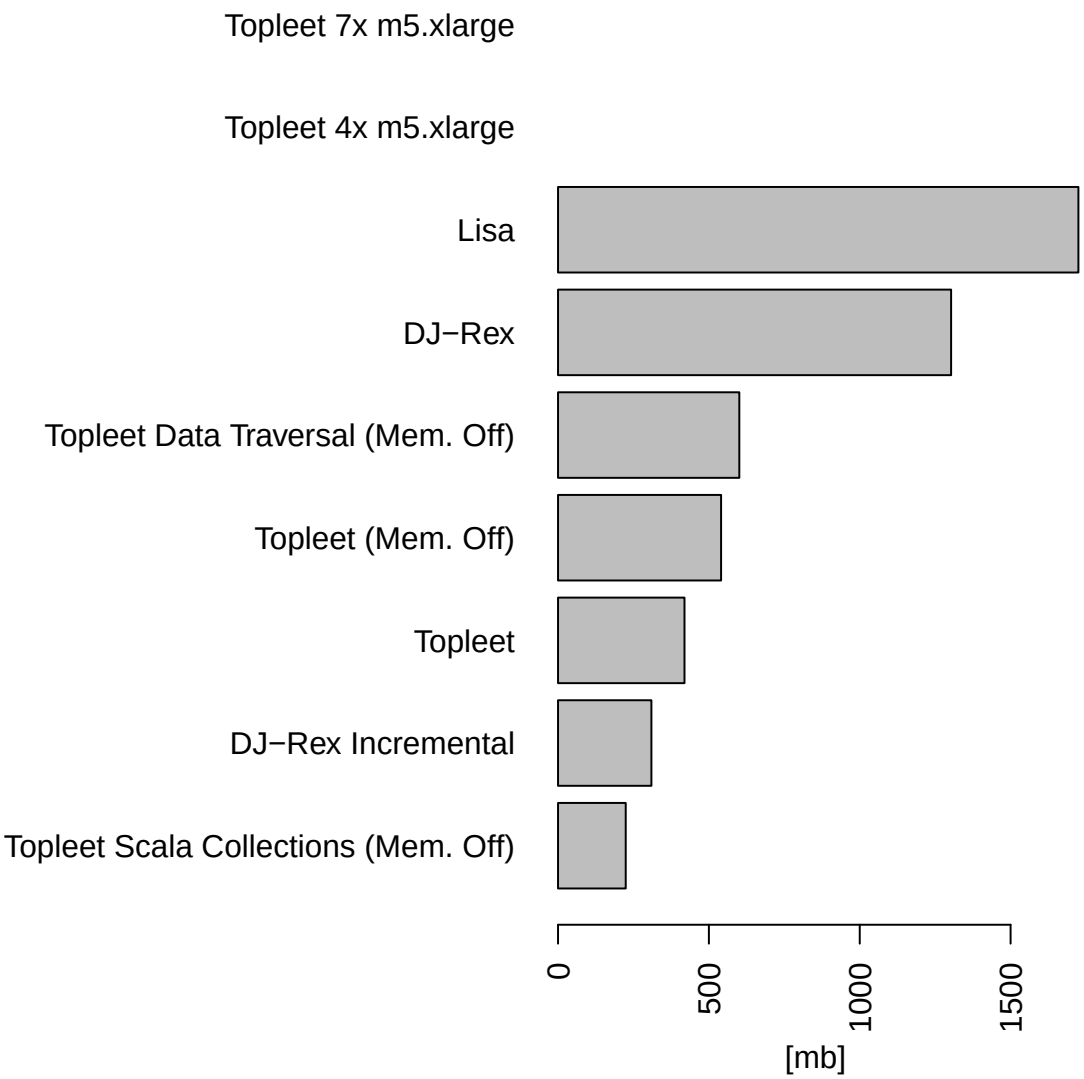


Figure 3.5: The average memory usage of a method, running on one of the 98 repositories, sorted by memory usage. The memory profile for distributed solutions is left blank.

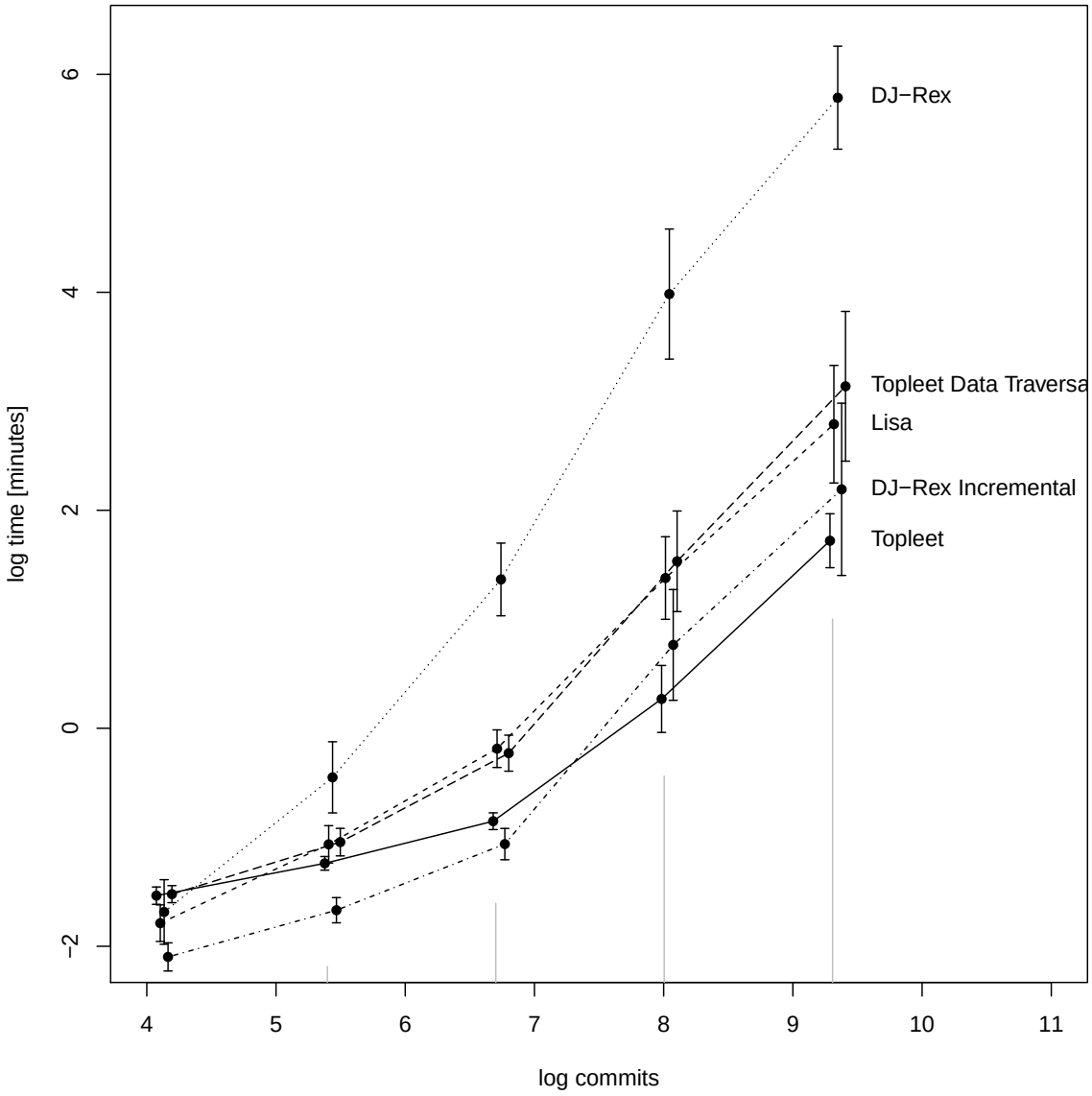


Figure 3.6: Time in minutes (**log-log scale**): Curves are shown with a small offset to prevent overlapping of error bars.

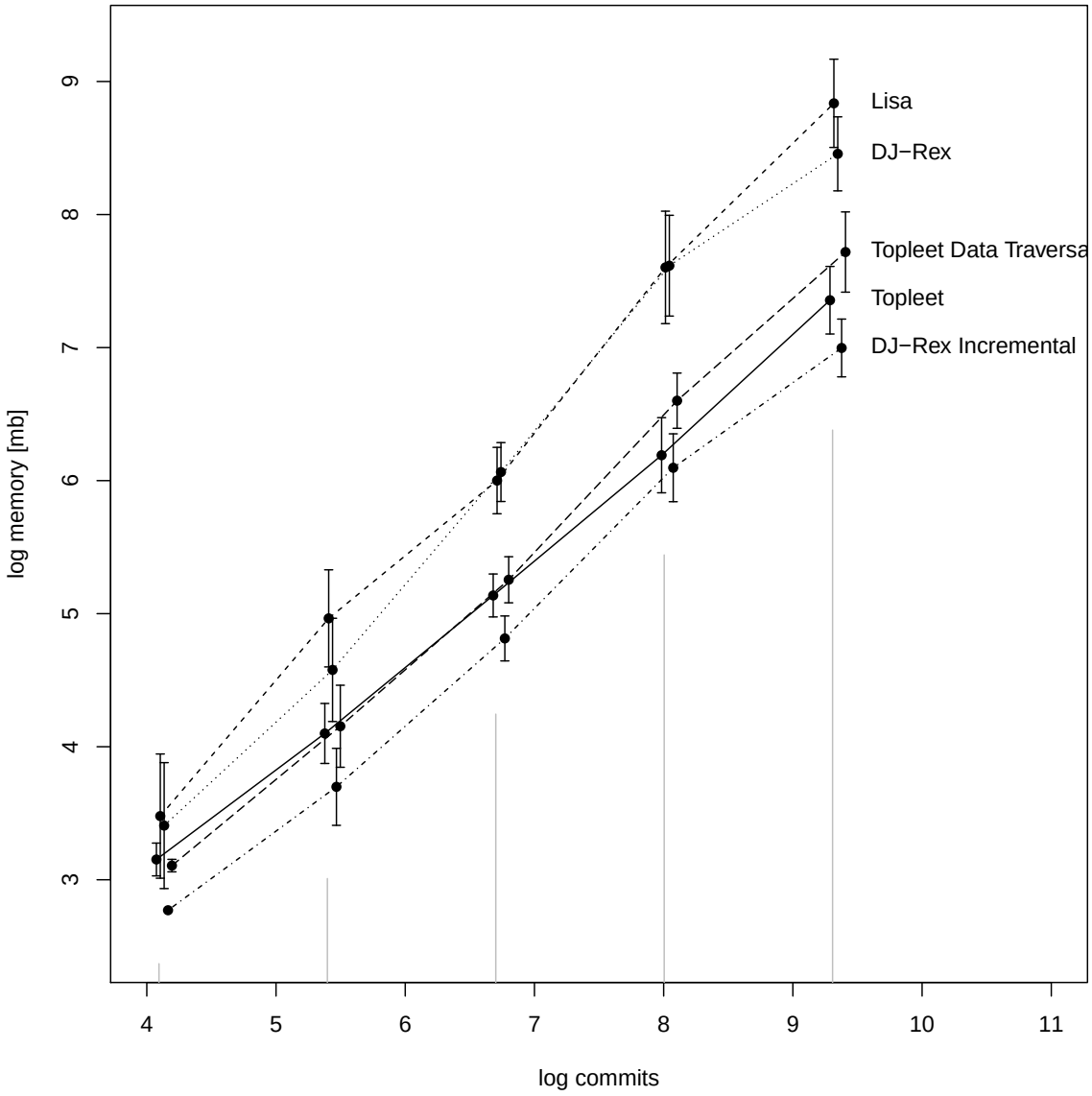


Figure 3.7: JVM Memory peak in mb (**log-log scale**):Curves are shown with a small offset to prevent overlapping of error bars.

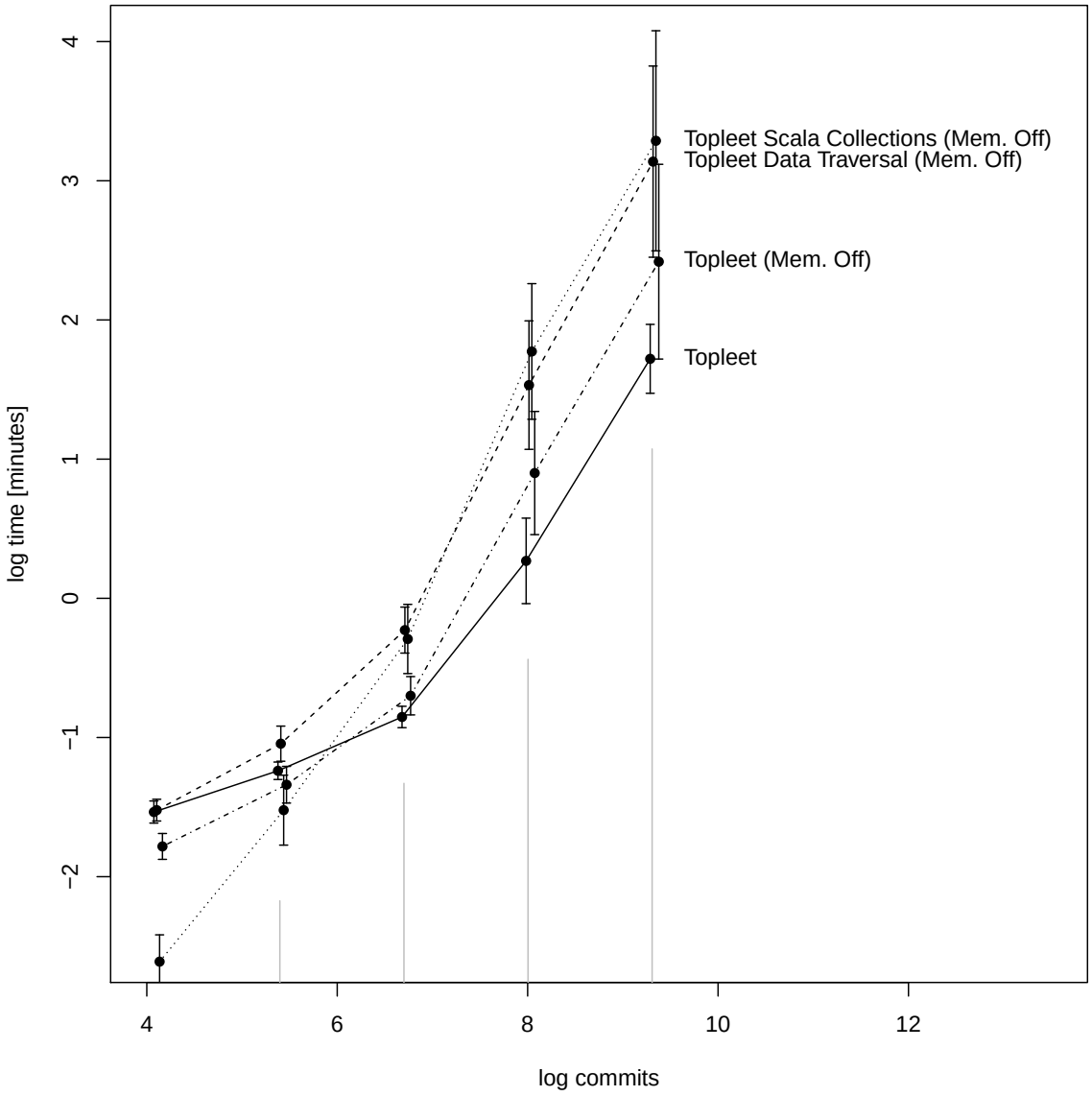


Figure 3.8: Time in minutes for different infrastructure solutions (**log-log scale**): Curves are shown with a small offset to prevent overlapping of error bars.

average 67.12 minutes) followed by Topleet Data Traversal without memoization (10.61), LISA (6.44), DJ-Rex Incremental (5.4) and Topleet (1.33). DJ-Rex, DJ-Rex Incremental and Topleet Data Traversal do not employ memoization. This appears to be beneficial for low commit count but hampers the performance on high commit counts when computations tend to reoccur – reflected by a bend in the average time at commit count 2990, for non-memoizing solutions.

3.5.7 Memory

The comparison of the maximum amount of memory used in the JVM is done analogue to the comparison of time and depicted in Figure 3.7 and Figure 3.5. We measure the memory peak for different repositories, minimizing the influence of garbage collection.

Memory usage increases with rising commit count. For the average memory peak over all repositories, DJ-Rex Incremental (0.32 GB) is sightly better than Topleet (0.44), both running on Apache Spark in local mode. Topleet Data Traversal Memoization Off (0.6) does not produce high overhead. DJ-Rex (1.34) and Lisa (1.81) both use the most memory.

3.5.8 Computation Infrastructure

We compare the application time using different Topleet infrastructure features in Figure 3.8. The Scala Collections API solution is best for low commit counts because it does not boot Apache Spark. For high commit counts, Topleet with memoization is the best (1.33 minutes on average), no memoization increases the time (5.23), followed by the data traversal variant (10.61) and the Scala collections realization (13.94).

3.5.9 Distribution

Increasing the number of core nodes in a distributed setup from 3 to 6 (i.e., Topleet 4x m5.xlarge and 7x m5.xlarge) changes the average time needed for repositories from 2.23 to 1.39 minutes. Figure 3.9 shows the improvement for the different commit buckets. For repositories with low commit counts, the improvement is low; adding hardware might even slow down the processing. For repositories with high commit count, the improvement is around 60%, for an 100% increase in hardware.

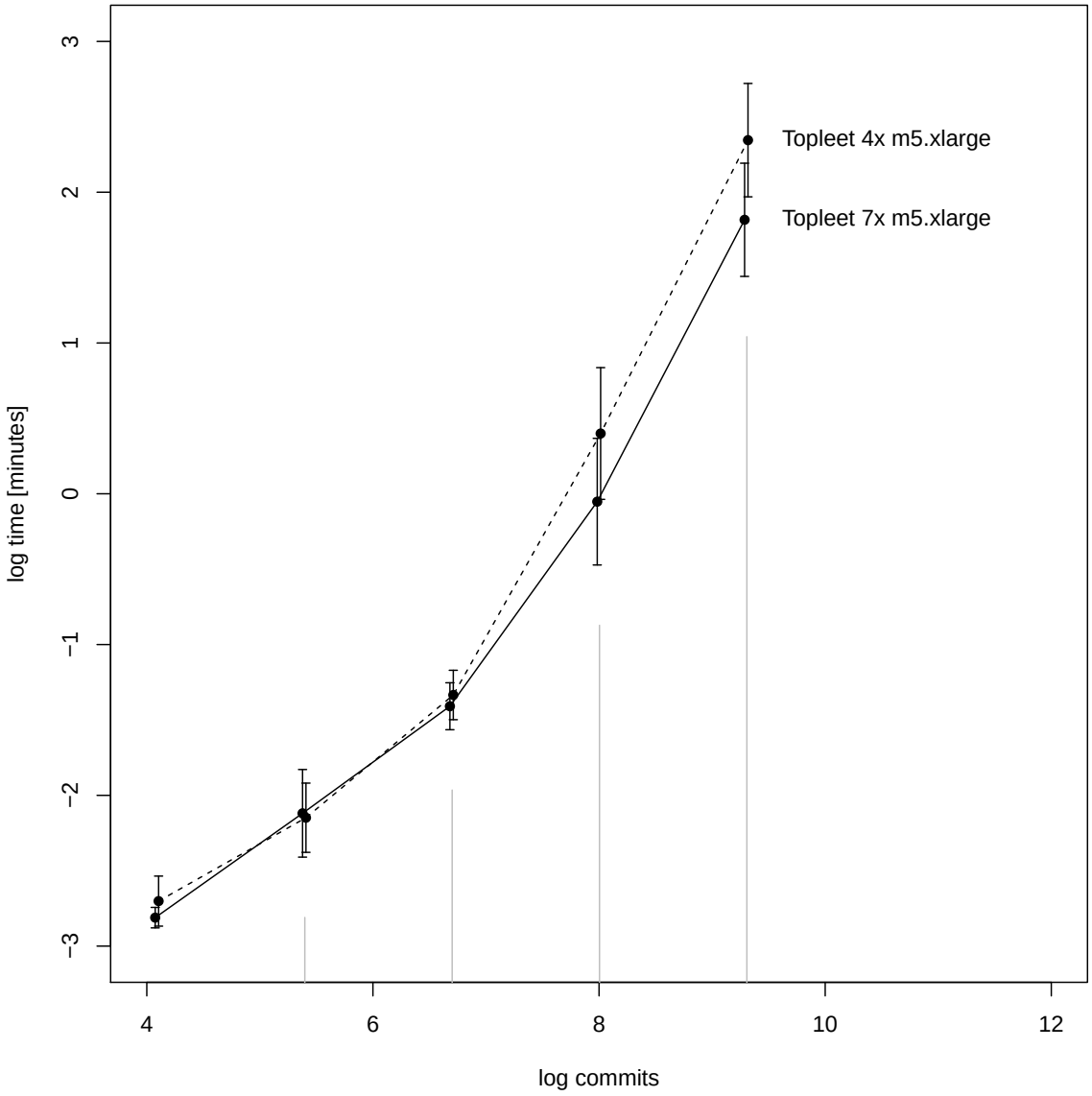


Figure 3.9: Distribution: The time running a solution on 7 cluster nodes (7x m5.xlarge) and 4 cluster nodes (4x m5.xlarge). Both solutions include one master node.

3.6 Conclusion

In this chapter, we cover our first technical challenge of scalable computation of abstractions of the revisions of a repository. We focus on incrementalizing map-reduce, motivated by previous occurrences of the scalability challenges in our working group, and by previous work on map-reduce in MSR/ESE by [SJAH09].

We propose a method to improve the scalability of functions that abstract over repositories with high revision count in a theoretically founded way. We use insights on abstract algebra and program incrementalization to define a core interface of high-order functions that compute scalable static abstractions of a repository with many revisions. We evaluate the scalability of our method by benchmarks, comparing a prototype with available competitors in MSR/ESE.

In the next chapter, we will switch to Datalog and use it to mine repositories with a very heterogeneous technology stack. Our placeholder involves a technology stack including divers artifact types and relationships in between the artifacts. We use Datalog since it allows a very modular definition of abstractions over heterogeneous repository content, facilitated by the declarative nature, not forced to any sequential order, to facilitate composition. This stand in direct contrast to non-declarative map-reduce, as we have used it in this chapter.

Chapter 4

Repository Mining with Datalog

4.1 Introduction

We start with describing the challenge of mining repositories with a heterogeneous technology stack, as pointed out in some of our previous work, e.g., in [RRH+20], and summarize the declarative solution proposed in this chapter.

4.1.1 The Heterogeneity Challenge

Typical studies that use repository data to understand software development need to understand a heterogeneous (or diverse) technology stack, too, that potentially manifests in a repository and its fragments [HK06, PML15, SBH+19]. For instance, empirical studies do not get rich abstractions of the source code for free (ASTs). Studies need to understand the code in its surrounding, including technological aspects, like the build system [MAH10, LPS11, GdCZ19], dependencies management [LPS11, SB15], various possible interoperating languages [BTL+13], infrastructure as code [OZR22, OZVR21], or MDE technology invoking code generation [ZS06]. Even when limiting following up analysis to a very specific mixture of technologies (e.g., to mobile apps [SPN+18] or graph query languages [SHL+19]), an initial understanding of the technology stack, used in a repository, is still needed as an inclusion/exclusion criteria. Defining abstractions over a heterogeneous technology stack can be complicated due to the flexibility of how technologies compose and interrelate (see work on documenting technologies and languages in software projects, which is motivated by such heterogeneity [FLV12, LV14, HLV17, HHL+17, Hei22, RRH+18, RRH+20]). Concrete analysis meets a challenge with such heterogeneous repositories.

To understand heterogeneous technology stacks in MSR/ESE, concrete studies often rely on monolithic functionality trying to consider all eventualities when processing the repository and its fragments. See examples like, [KG11, SPN+18], and an overview of tools used [CSS13]. Unlike the underlying technology stack, such monolithic mining functionality cannot be composed in a straightforward manner. We are aware of one exception, where a method aims at language independent analysis (LISA) [APPG19]. Such exception motivates the need for good solutions that facilitate analysis of heterogeneous situations.

4.1.2 A Declarative Solution using Datalog

Since the technology stack of a repository can be composed flexibly, we assume that mining logic needs to be composed flexibly too. This calls for modular solutions.

Declarative logic programming follows this ambition, describing a program without focusing on its control flow. *Declarative* refers to a language which is less sensitive to details on the control flow and *logic* refers to an execution that comes close to a proof-derivation (see the general discussion of classifications in [Läm]). Datalog is an instance of a declarative and logic programming language, which allows defining programs in terms of modular rules [DEGV01, GHLZ13].

In this chapter, we show how to use Datalog to understand a heterogeneous technology stack. We transfer existing ideas, previously presented in the context of architecture recovery, source code querying, and static program analysis, from an analysis of a homogeneous technology stack, limited to a single programming language, to an analysis of a heterogeneous technology stack, involving very different artifact types and languages. While the past work assumes well-defined abstractions of the source code to be available upfront to analysis, we focus on understanding the full surrounding of a complex technology during the analysis. Our placeholder of studying MDE technology is sufficiently complex. Understanding MDE technology requires understanding a heterogeneous mixture of other technologies first, related to the build system, XML, Java and different MDE processes (including code generation).

We propose a method to improve the definition of functions that abstract over a repository with a heterogeneous technology stack, by using concepts from declarative logic programming (in particular Datalog [DEGV01, GHLZ13]) and combining them with ideas on megamodeling and linguistic architecture (see [FLV12, LV14, HLV17, HHL$^+$17, Hei22, RRH$^+$18, RRH$^+$20]). We reproduce existing ideas on declarative logic programming with languages close to Datalog, coming from architecture recovery [MMW02, MT01, TM03], source code querying [HVdM06], and static program analysis [BS09, SBEV18], and transfer them from the analysis of a homogeneous to a heterogeneous technology stack. In particular, we facilitate understanding a complex technology stack in a repository by a bottom-up, step-by-step, and modular classification of the repository, its fragments, and the involved technologies, using Datalog rules. The method finally leads to a non-trivial understanding of the repository and the involved technologies in the large. Results conform to schemata from previous work on megamodeling and linguistic architecture, which are developed to document how complex technologies and languages manifest in software projects.

Our method facilitates modularity to fight the heterogeneity present in a technology stack. We use modular rules to infer classifications from other existing classifications. Rules produce classifications that conform to ideas on megamodeling and linguistic architecture, ideas that have proven to be well suited to describe the complex composition of technologies in previous work [FLV12, LV14, HLV17, HHL$^+$17, Hei22, RRH$^+$18, RRH$^+$20]. Starting with basic classifications, e.g., of the revision's resources, our method uses modular Datalog rules that trigger more complex classifications, e.g., of the build system, Java, XML, or MDE technology. Finally, we may apply overall complex classification of technology patterns which might be part of the repository. Our classifications can be read like proof-derivations, which helps to

trace interrelations between modular rules, separately classifying independent parts of technologies.

A novel aspect of our method, compared to previous work on architecture recovery, source code querying, and static analysis, is a flexible way of integrating heterogeneous repository content into the Datalog reasoning. We provide an interface for plugging pure high-order functions, typically parametrized by query terms, which organize a traversal of the repository content. The traversal is initialized at the repository root and navigates to arbitrary fragments by function application. This may include fragments of different artifact types, like the file-system, XML or Java. We use uninterpreted function terms for the identification of such fragments. This allows a fluent mining process while other methods need to go an indirection, e.g., over databases [HVdM06]. Our flexibility in data integration helps to work around limitation imposed by data size, heterogeneous schemata, and may be used to avoid operating on complete ASTs or similar structures (as in, e.g., [Roo11, RLP13, AA17, SME+17, APPG19, HVdM06]).

Opposed to the map-reduce analysis style (e.g., used in [SJAH09, HL20]), the analysis with Datalog takes an important limitation, enabling recursive queries in native manner. Recursion is an essential part of advanced (program) analysis [BS09, SBEV18, GHK+19, HVdM06]. It is also mandatory in recent empirical contributions in ESE/MSR, e.g., to chase transitive dependencies [ZMDR22].

We provide a prove-of-concept of such method in a case study. We apply the previously reoccurring ideas from architecture recovery [MMW02, MT01, TM03], source code querying [HVdM06], and static program analysis [BS09, SBEV18], to a novel, heterogeneous context, studying the Eclipse Modeling Framework (EMF). EMF is a very heterogeneous technology combining XML, Java, OSGI and various build systems in its application [SBMP08]. This heterogeneous context requires a more flexible access to the repository and its fragments. The case study defines and runs the Datalog rules to mine EMF technology patterns. We step-by-step classify different artifacts, part of EMF, find relationships (code vs. model vs. generator), and finally do a high-level classification that detects specific EMF technology usage patterns. We separate the rules for all technologies, and finally apply rules that classify relationships. We apply the mining to GitHub repositories.

4.1.3 Properties of Datalog

Datalog share the following interesting properties, that makes us consider it as an evident alternative (compared to map-reduce or SQL) to produce such bottom-up, step-by-step classification:

- Datalog is a subset of Prolog which can benefit from more **efficient algorithms** to evaluation, for instance, it can be incrementalized [GHLZ13].
- Datalog's bottom-up evaluation, using rules to infer facts from existing facts comes close to our idea of a **step-by-step, bottom-up, and modular classification** of the repository and its fragments. We infer classifications from existing classifications: Stating with the most basic classifications (e.g., of the revision's resources) rules trigger more complex classifications (of the build

system, Java, XML or MDE technology); which finally leads to an overall complex classification of the revision.

- Datalog uses **modular** rules, insensitive to order. We use this to decompose our classification effort for the very heterogeneous artifact types. Possibly, we may be able to reuse such logic in different MSR/ESE studies. This increases the usability for different user groups.

- Our rules operate on the classifications of repositories and its fragments. Some classifications may trigger other rules that contribute additional classifications. This calls for **recursion**. Typical query languages, like SQL or map-reduce, do not allow such flexibility in the **interaction** of queries (rules) triggering each other. Datalog enabling recursive queries natively [GHLZ13].

- Datalog facts (the classifications) can be structured to conform to a schema which resembles parts of our previous work on modeling in the large, referred to as **megamodeling or linguistic architecture** [FLV12, LV14, HLV17, HHL+17, Hei22, RRH+18, RRH+20]. We build on this previous knowledge on how to classify complex technology to organize the extracted data systematically.

- There is recent work on **static analysis** with Datalog that realizes, e.g., points-to-analysis [BS09] or the analysis of control flow [SBEV18]. Both are relevant for an in-depth understanding of repositories and may be reused for our method. We show usage close to such complexity, finding relations between artifacts and fragments thereof (providing classifications in terms of tuples).

4.1.4 A Small Example

To give the reader an idea of our method, consider the following trivial Datalog rule drawn from an upcoming case study. The rule shows how to start from a basic classification as file and XML, and derive a higher-level classification as metamodel.

```
1 elementOf(?x, Metamodel) :-
2     manifestsAs(?x, File),
3     elementOf(?x, XML),
4     "ecore" = Extension(?x).
```

Listing 4.1: Sample rule classifying Ecore files.

The body of the rule (i.e., the condition right to ':-', line 2-4) quantifies over artifacts `?x` that are files with extension 'ecore' and readily known to be of 'type' (language) *XML*. For each such artifact, the head of the rule (left to ':-', line 1) states that the artifact is also of 'type' (language) 'Metamodel'. Thus, the rule infers new facts for artifacts to be classified as (Ecore) metamodels.

Notice that this rule does not give an explicit statement on the control flow (declarative style) and the rule is modular in a sense that it can easily be accomplished by other rules for composing complex classifications. We later show how to complement such rule by a series of other rules, used to refine the classification of very heterogeneous artifacts in a repository step-by-step.

An important part of our method is an extensible suite of accessor primitives for interacting with formats and structures part of repositories, such as, *Java*, *XML*, and the file system (the folder structure). One example can be found in the previous rule, where an accessor primitive `Extension` (highlighted by underlining) queries the repository for a file extension. We consider this as a very specific integration problem, allowing the exchange of data between the repository and the Datalog engine. Our method favors not going an indirection over databases, but navigating the repository and its fragments on-demand by high-order functions.

The dataset for the case study and the implementation of the prototype are available online[1].

4.1.5 Summary of this Chapter's Contributions

- Our technical contribution is a reproduction of previously reoccurring methods on using Datalog in architecture recovery, source code querying and static analysis, transferring it to a novel application case of analyzing repositories with a heterogeneous technology stack. We provide integration mechanisms for accessing very heterogeneous repository content in a modular and declarative way. We combine this with previous research on megamodeling and linguistic architecture to represent the extracted data.

- We execute a case study as a proof-of-concept, to evaluate our method. We illustrate the capability of the method in recovering a catalog of complex *EMF* technology pattern in 5759 GitHub repositories. This contribution overlaps, in parts, with the thesis by Marcel Heinz, who developed the catalog [Hei22]. The catalog and the corresponding empirical results, are associated with Heinz, while the Datalog method used for mining is associated with this thesis.

4.1.6 Summary of the Delta to the Publication

The following items describe the delta of this thesis to previous publications.

- In [HHL18], the method, the pattern catalog, and the case study on EMF are presented. This publication founds on the Apache Jena syntax for describing the analysis (a very specific Datalog dialect that appears in the domain of semantic web).

- This thesis differs in that it translates the syntax into conventional Datalog. Rephrasing allows a more direct discussion of core Datalog features. This thesis also improves the presentation of the case study results (e.g., in terms of better visualizations and in-depth examination of the data). We also add an in-depth introduction to the background on Datalog.

- We improve the related work section respective architecture recovery [MMW02, MT01, TM03], source code querying [HVdM06], and static program analysis [BS09, SBEV18].

[1] `https://github.com/softlang/qegal`

4.1.7 Micro road-map of this Chapter

This chapter follows the metamodel defined in the introduction (Sec. 1.6). In particular,

Sec. 4.2 begins with a motivation, describing a running example on mining a code generation pattern from a repository. The example is used throughout this chapter and is part of the final case study on EMF.

Sec. 4.3 introduces the background on Datalog and the abstract pattern catalog used as a blueprint for the case study. To avoid toy examples, the background section illustrates general Datalog concepts solving the running example (mining code generation patterns from a repository). The term 'background' is used in a modest sense, as this section also contains a significant amount of original content.

Sec. 4.4 accomplishes the presentation of our method, focusing on the integration of very heterogeneous revision content into the Datalog engine, and on referencing mechanisms used to identify repository content.

Sec. 4.5 evaluates the method by a prototypical realization of a complex case study (on mining EMF usage from GitHub).

4.2 Motivation

This section introduces a running example on mining a code generation pattern. Mining code generation of EMF will later be part of the case study (Sec. 4.5). It is the placeholder we use to present our method.

Why to Detect Code Generation? MDE [Béz05b, Béz05a, Sch06] has the goal to increase the quality and productivity in software development by the usage of models, metamodels, model transformations, and model comparisons (note the ambiguity of models in this thesis). In essence, developers try to work with (high-level) models instead of low-level code (also considered as model) whenever possible. Often this is done to abstract from implementation-specific details, that are finally derived from the high-level models using model transformations. Models may also be used in the communication between developers, like often done using UML. Adoption of MDE in software development is still subject to empirical research and the benefits are discussed controversially, e.g., see work on EMF related technology in [KMK+15], or work on the relation between UML and software quality (defects) examined in [RHC+19, RHH+17].

Code generation is a very central aspect to such MDE technology, which makes models executable. EMF generates code from class diagrams; UML may be translated into executable bits of code. However, what is typically very relevant during software development, is if there is a correspondence between the UML/EMF model and the code. If both do not correspond, the model might be an outdated leftover, subject to earlier development phases. If taking the existence of such a model as empirical evidence for the active application of MDE, one might be mistaken. Hence, certain MSE/ESE analysis may have an interest in the detection of up-to-date code generation.

Furthermore, we assume that code generation is a good placeholder, closely related to a series of problems that appear during the technical analysis of very heterogeneous repository content involving heterogeneous technologies. First, we need to find relationships between artifacts and not just examine isolated artifacts. Second, we need to examine different types of artifacts (model and code). Third, we might need to examine the correspondence, having a recursive look at the structure of the different artifact types.

In the following, we introduce a concrete running example as a placeholder, but we assume that the discussion of this chapter can be transferred to other closely related questions sharing the same characteristics.

Detecting Code Generation Figure 4.1 shows the typical artifacts involved in a basic code generation (and fragments thereof). We also include fragments, since we want to point out how nesting structures may be examined by recursion.

First, we have two different artifact types that we need to examine with an internal structure. The metamodel M to the right, i.e., a simplified class diagram, contains nested diagram elements that reflect the classes A, B and C (the fragments of the metamodel). Java package P to the left of the figure shows the nested Java classes A, B and C (the fragments of the package).

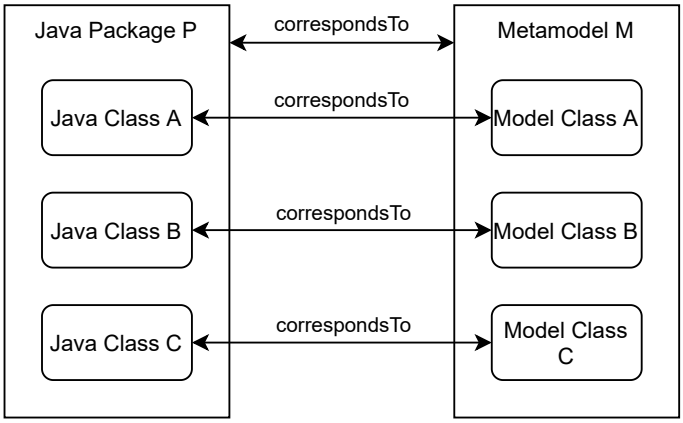


Figure 4.1: Placeholder Technology Pattern: Correspondence of a metamodels (right) and generated Java code (left).

Next to the structure of the different artifact types, the plot shows correspondence relationships that one might assume when facing code generation (depicted as arrows). The Java package P corresponds to the metamodel M and each Java class corresponds to a metamodel class. Such correspondence is only explicit at the moment of running the generation. Rerunning the generation might be an option to record this correspondence while being produced. However, reproducing the exact application of specific technology in the development environment, such as code generators, on heterogeneous repositories, is often too complex.

In practice, such generation can also be detected by examining hints on the past execution of the generation process that manifest in the repository. The correspondence might be inferred by employing naming conventions, unique identifiers, artifacts (like generator scripts), or by checking (deep) structural correspondence of metamodel and package. This is an approximation of the development runtime behavior somewhat similar to the approximation of program runtime by points-to-analysis.

Different empirical questions might depend on such correspondence, as previously sketched. In this chapter, we focus on the technical contribution of mining with Datalog. The case study will include a preliminary empirical discussion of the data on EMF.

4.3 Background

This section introduces background on Datalog, background on megamodeling and linguistic architecture, and an EMF pattern catalog. The catalog was developed by our working group (in particular Marcel Heinz) in the context of megamodeling and linguistic architecture.

4.3.1 Declarative Logic Programming in Datalog

After early research in the eighties and early nineties, Datalog almost disappeared from research. However, due to technical advances, recent work from different domains picks up such research again [HGL11, GHLZ13]. In particular, work covers architecture recovery [MMW02, MT01, TM03], information extraction [SDNR07], source code querying [HVdM06], static program analysis [LWL+05, EKKM08, BS09, SB10, SBEV18, Sza21] and data integration [Len02, FKMP05, GKIT07]. Some of those methods, e.g., by Mens et al. and Tourwé et al., use inference rules that look identical to ours. We consider such previous efforts as a motivation for our work. We complement the original methods putting a focus on understanding repositories with a very heterogeneous technology stack.

Datalog is well-known, so we only refer to a recent survey for a general audience [GHLZ13]. However, in the following, we will give the background on Datalog to make this thesis self-contained. The survey inspires our discussion.

The running example that we use in this section, and the corresponding Datalog solution to mine correspondence, is original to this work.

Defining the 'what' and not the 'how'

The essence of declarative programming is to define the 'what', and not the 'how'. We start with some informal statements on 'what' to find in a repository. We want to avoid including too many details on 'how' to find it, e.g., by instructing the program on which SQL query to run first; or how to join.

For the running example, mining should assure the following. Most statements can be read as classification of the repository and its fragments.

- The first artifact is classified as metamodel.
- The second artifact is classified as Java package.
- There is a fragment of the metamodel that can be classified as its identification.
- There is a fragment of the Java package that can be classified as its identification.
- The identification of metamodel and package are the same.

Facts The following Datalog *facts* describe the previous statements on the code generation. In particular, the facts describe a scenario where a generation process uses the metamodel at path `'repository:/model.ecore'` to generate the Java package located at path `'repository:/src/package'`.

```
1 manifestsAs('repository:/src/package', Folder).
2 manifestsAs('repository:/model.ecore', File).
3
4 elementOf('repository:/src/package', JavaPackage).
5 elementOf('repository:/model.ecore', Metamodel).
6
7 id('repository:/src/package','http://my.id.org').
8 id('repository:/src/model.ecore','http://my.id.org').
```

Listing 4.2: Facts related to code generation in Datalog syntax.

Datalog facts begin with a predicate symbol, such as, `manifestsAs`, `elemmentOf`, or `id`, followed by a list of constants in braces. In our example, artifacts that are fragments of the revision are referenced by path constants (starting on `repository:/`). Other constants, like `File`, `Folder`, `JavaPackage`, and `Metamodel`, are used as classifier (that do not materialize in the revision). The constant `'http://my.id.org'` is a string literal and represents the identification of the metamodel and Java package. Literals can be considered as parts of the raw content of a revision. In the ongoing discussion (Sec. 4.4.2), we will add details on constants and explain the referencing mechanisms for revision content. For now, the details can be skipped.

The facts mirror the content of a repository: One of the artifacts is a `Folder` and a `JavaPackage`; the other artifact is a `File` and a `Metamodel`. Such classifications are given by the predicate `manifestsAs` for the content type and `elementOf` for further language classifications. We will also have a closer look at this schema used to mirror the repository content as facts in Sec. 4.3.2 (such distinction is motivated by previous knowledge on megamodeling [FLV12, LV14, HLV17, HHL+17, Hei22, RRH+18, RRH+20]). In the last two lines of the code (7-8), a predicate `id` assigns both artifacts to the same identification.

For now, we assume that these facts are part of the set of previously known classifications, integrated upfront to our analysis; hence, we can directly start with Datalog reasoning. We will add details on how to integrate such facts from a repository in Sec. 4.4.1.

Pattern Matching A central aspect of working with Datalog facts is pattern matching. Pattern matching can be compared to running a basic query against the facts (close to regular SQL or map-reduce queries).

Suppose we are interested in facts matching `elementOf(?x, Metamodel)` where $?x$ is a placeholder. This corresponds to querying artifacts that are metamodels. According to our previous list of facts, the assignment of placeholder $?x$ is `{'repository:/model.ecore'}`. Such query can also be viewed as a simple filter operation on the set of facts defined by the predicate `elementOf`. The second entry of the tuple needs to be equals to the `Metamodel` classifier. It can also be implemented as a basic SQL query when considering `elementOf` to be represented as a table.

Now suppose we are querying for a more complex pattern. We aim to find the identification of the metamodel artifact by writing `elementOf(?x, Metamodel), id(?x,?y)` where $?x$ and $?y$ are placeholders. When looking at the facts, we will only find one possible assignment for $?x$ and $?y$ which is `{('repository:/model.ecore', 'http://my.id.org')}`. Such query can be re-

alized in terms of a select/project/join strategy on all facts (possible in SQL or map-reduce).

Rules However, pattern matching is not sufficient to implement complex cascades of step-by-step, bottom-up and modular classifications of the repository and its fragments.

Datalog allows rules that infer new facts from existing facts. We use it to infer new classifications from existing classifications. The following rule infers a correspondence that classifies the code generation that we search for. It adds a new fact `correspondsTo` to the data.

```
1 correspondsTo(?p,?m) :-
2     manifestsAs(?p, Folder), manifestsAs(?m, File),
3     elementOf(?p, JavaPackage), elementOf(?m, Metamodel),
4     id(?p, ?id), id(?m, ?id).
```

Listing 4.3: A declarative solution finding a code generation pattern.

The body of the rule (i.e., the condition right to ':-') is a pattern matched against existing facts. In order to detect correspondences, the placeholders $?p$, $?m$ and $?id$ are matched. Placeholder $?p$ needs to manifests as `Folder`, $?m$ as `File`, $?p$ is classified as `JavaPackage`, $?m$ as `Metamodel`, and both have the same id $?id$. There is a single match, i.e., `{('repository:/src/package', 'repository:/model.ecore', 'http://my.id.org')}` for $?p$, $?m$, $?id$.

The head of the rule (left to ':-') now adds a new fact using the placeholder assignment. It produces the fact shown in Listing 4.4.

```
1 correspondsTo('repository:/src/package','repository:/model.ecore').
```

Listing 4.4: A new fact is created.

This primitive form of a rule can be understood as a named query. The rule's body defines the query by pattern matching, and the rule's head names the result. However, the difference to regular SQL and map-reduce comes with the interaction of rules. Comparable to queries reusing other queries, rules may infer facts that trigger other rules. However, in some cases, Datalog does not require a strict order of queries using each other, and it allows recursion.

Datalog allows us to focus on the produced facts (our classifications) that follow a well-understood schema based on previous knowledge on megamodeling and linguistic architecture. It does not force us to name queries and organize their order of execution. Such a mechanism to classification is modular if we agree on the schema. We can add and remove rules that step-by-step refine the classification of the repository. Possibly, we may classify recursively.

Recursion Adding a new fact to the existing facts may potentially trigger another (or the same) rule to mach again. Hence, the process of matching and adding facts may repeat until a fixpoint is reached, and no more facts can be added to the data.

Such recursion can be examined on top of a *precedence graph*, where all predicates involved in a Datalog program are depicted as nodes. In this graph, directed edges between predicates are inserted if used in the body and head of the rule, respectively. Figure 4.2 shows such graph for the predicates of the correspondence rule

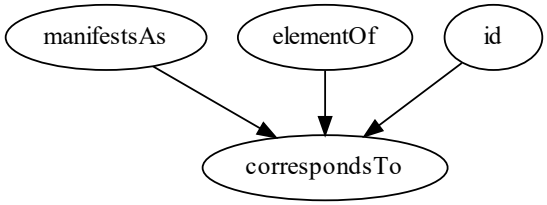


Figure 4.2: Precedence graph

(Listing 4.3). New facts for predicate `manifestsAs`, `elementOf`, and `id` are used to infer new facts for predicate `correspondsTo`. Adding more rules may blow up such a graph; however, as long as we face an acyclic graph, the underlying logic is free of recursion (we do not need a fixpoint computation). A basic project/select/join strategy (regular map-reduce or SQL query), computing the predicates in the order of a topological sorting of the precedence graph, suffices to get an equivalent to the Datalog program.

We will now extend this by a recursive rule for dealing with deep correspondence. As shown in the diagram that illustrates code generation (Figure 4.1), we assume that there are nested components of a package and metamodel that may correspond to each other. For illustration, we add new `partOf` facts to the existing facts (manually) that describe the structure of the metamodel and the Java package (containing classes A, B and C and corresponding identifications).

```
1 partOf('repository:/src/package/ClassA.java','repository:/src/package').
2 partOf('repository:/src/package/ClassB.java','repository:/src/package').
3 partOf('repository:/src/package/ClassC.java','repository:/src/package').
4
5 id('repository:/src/package/ClassA.java','ClassA').
6 id('repository:/src/package/ClassB.java','ClassB').
7 id('repository:/src/package/ClassC.java','ClassC').
8
9 partOf('repository:/model.ecore#ClassA','repository:/model.ecore').
10 partOf('repository:/model.ecore#ClassB','repository:/model.ecore').
11 partOf('repository:/model.ecore#ClassC','repository:/model.ecore').
12
13 id('repository:/model.ecore#ClassA','ClassA').
14 id('repository:/model.ecore#ClassB','ClassB').
15 id('repository:/model.ecore#ClassC','ClassC').
```

Listing 4.5: Nested artifacts

We also add a second rule for inferring nested correspondence. It examines the parts of an existing correspondence relation and checks if parts with equal identification exist. It creates a new correspondence fact for them, if this is the case.

```
1 correspondsTo(?part_x,?part_y) :-
2     correspondsTo(?x,?y),
3     partOf(?part_x, ?x), partOf(?part_y, ?y),
```

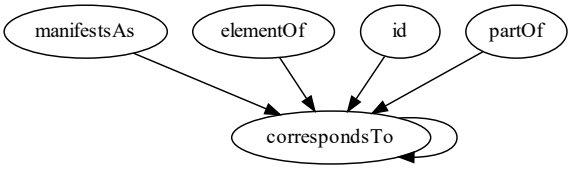


Figure 4.3: Precedence graph with recursion

```
4    id(?part_x, ?id), id(?part_y, ?id).
```

Listing 4.6: A recursive definition of (deep) correspondence

The new precedence graph (Figure 4.3), including the rule from Listing 4.3 and Listing 4.6, shows the new predicate `partOf` and a self reference for the predicate `correspondTo`. Hence, `correspondTo` may be invoked recursively, adding facts, while checking the nested parts across multiple layers. We limit the example to the class level, but the program may potentially recur into deeper layers, examining structural properties of classes. The following facts are added by running the new rule.

```
1 correspondsTo('repository:/src/package/ClassA.java','repository:/model.ecore#ClassA').
2 correspondsTo('repository:/src/package/ClassB.java','repository:/model.ecore#ClassB').
3 correspondsTo('repository:/src/package/ClassC.java','repository:/model.ecore#ClassC').
```

Listing 4.7: Facts infered by the recursive definition of (deep) correspondence

As previous mentioned, Datalog provides an intuitive and modular way to write such classification. Realizing the same kind of recursive classifications in map-reduce or SQL is not straightforward.

Stratified Negation So far, we have an answer for the existence of a correspondence relation, but often we are also interested in nonexistence.

In the following, we add a rule inferring whether there is a missing structural correspondence where a higher level correspondence demands for it. This may answer, for instance, if a metamodel class is missing a corresponding Java class. In our running example, we will refer to this new insight as fact `notUpToDate`. This comes close to one of the patters we search for in the case study. We also need to add an intermediate predicated `deepCorrespondsTo` for checking deep structural correspondence of two parts.

```
1 deepCorrespondsTo(?x,?part_x,?y,?part_y) :-
2     partOf(?part_x, ?x), partOf(?part_y, ?y),
3     correspondsTo(?x,?y), correspondsTo(?part_x,?part_y).
4
5 notUpToDate(?y, ?part_y) :-
6     correspondsTo(?x,?y),
7     partOf(?part_y, ?y),
8     not deepCorrespondsTo(?x,?part_x,?y,?part_y).
```

Listing 4.8: Use of negation in a rule for checking the absence of deep structural correspondence

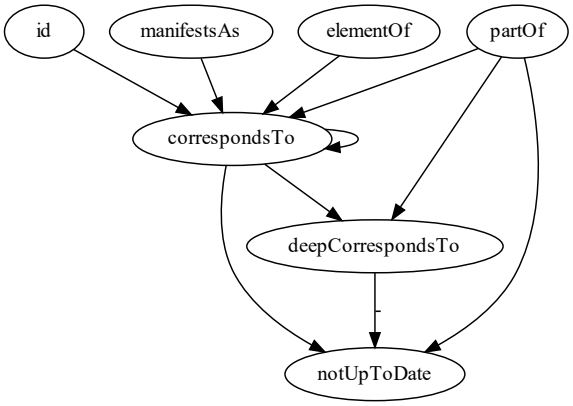


Figure 4.4: Precedence graph with recursion and negation

This rule for producing `notUpToDate` classifications only applies if we know for sure that there is a correspondence between two artifacts, $?x$, and $?y$. It examines the parts of placeholder $?y$ and checks whether there is a deep structural correspondence `deepCorrespondsTo` missing for the $?part_y$ using `not`. A symmetric definition for the missing parts of $?x$ can be given in analogy.

Negation changes how the basic rule-based inference mechanism works. Datalog will stop being monotonous, i.e., adding new facts may have the effect of invalidating previous facts and thereby invalidate the application of rules.

Dealing with removed facts caused by negation can be circumvented if the computation of predicates in the precedence graph is sorted along the topology of strongly connected components (called strata). This assures that, during the computation of higher strata, lower strata will be up-to-date and reasoning will not face situations where previously inferred facts are invalidated.

However, there are exception where this strategy does not apply. For the following two rules, `p :- not q.` and `q :- not p.`, no unique answer can be given. This problem is typically resolved by constraining negation to not appear in a recursion. This can be implemented as a static check on the precedence graph[2].

In Figure 4.4, we see a precedence graph for our example using negation (Listing 4.8). This graph does not include a negation in a cycle. The predicates can be computed in the order: `id`, `manifestsAs`, `elementOf`, `partOf`, `correspondsTo`, `deepCorrespondsTo`, and `notUpToDate`.

Stratified Aggregation Stratified aggregation follows the same principle as the stratified negation. Aggregation is expressed in the head of a rule. The following example shows how to sum up the classes that are not up-to-date in an artifact or

[2]An interesting aspect is that our analysis of deep non-correspondence also avoids this recursive use of negation. From a static perspective, this is correct, but if considering the structure of the artifacts, which is typically a tree, such limited use of negation is not necessary. The tree structure guarantees the order, comparable to stratification. We did not examine such issues in depth, but we want to point out that this does not limit analysis too much.

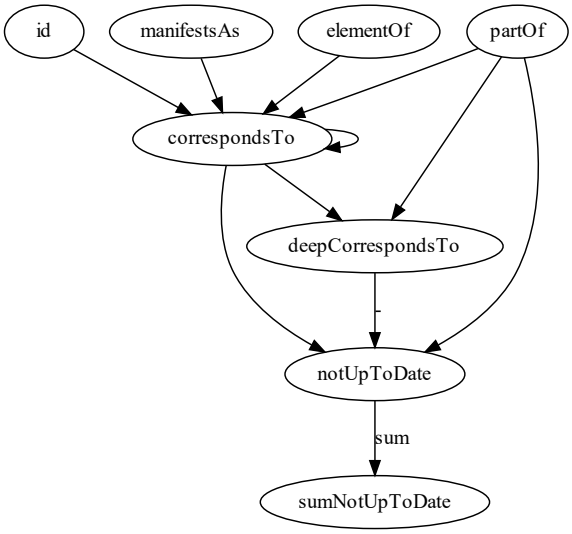


Figure 4.5: Precedence graph with recursion, negation and aggregation

part thereof. The precedence graph can be found in Figure 4.5.

```
1 sumNotUpToDate(?p, sum<?c>) :- notUpToData(?p,?c).
```

Listing 4.9: Use of negation in a rule for checking the absence of deep structural correspondence

Advanced Datalog Concepts Other concepts of Datalog include: components, i.e., some sort of template mechanism (e.g., provided by Souffle[3]); uninterpreted function terms (functors) for composing data types like lists; infinite relations for realizing custom functions. Our method also uses (uninterpreted) functions and infinite relations. We will explain them in greater detail when needed in Sec. 4.4,

4.3.2 Megamodeling and Linguistic Architecture

> The content on megamodeling and linguistic architecture, and the pattern catalog, including the corresponding empirical results, **are not part of the contributions of this thesis**. The pattern catalog was developed by Marcel Heinz. The empirical results of the case study on the EMF pattern catalog are associated with Marcel Heinz. We summarize the content here to provide this necessary background. The method used to mine the pattern catalog is a contribution of this thesis.

One of our long-term research objectives, present in the previous work of our working group, is to apply megamodeling (see [BJV04, BJRV05]) to the problem of documenting software-technology usage in software projects. In our previous

[3] `https://github.com/souffle-lang/souffle`

work [FLV12, LV14, HLV17, HHL+17, Hei22, RRH+18, RRH+20], we focused on case studies, basic aspects of language support, some forms of verification of a megamodel to correspond to a proper system abstraction, the axiomatization of the involved megamodeling expressiveness, the methodology for discovering megamodels, and surveying related concepts in the literature. We also use the term (models of) 'linguistic architecture' for such megamodels. We can consider this as a form of an abstraction of a repository.

From Megamodel and Linguistic Architecture to Datalog

In this thesis, we rely on the aspect of megamodeling and linguistic architecture to document heterogeneous language and technology occurrence in software projects. It serves us in 'structuring' the data that we extract from a repository (potentially with a heterogeneous technology stack) by a well-motivated schema. We employ the previous axioms and schemata, presented in the context of megamodeling and linguistic architecture, in a mining- (or reverse engineering-) oriented view, using closely related Datalog rules as instructions for the declarative mining of such abstraction. The extracted data, following such a schema, can be seen as a unified abstraction of the repository, focusing on languages and technologies.

We reuse the idea to classify artifact in terms of their structure, the used languages, and the technologies. We have presented such classifications in the previous fact in terms of:

- `manifestsAs` used to classify different kinds of Artifacts,
- `elementOf` used to classify the language,
- and `partOf` or `correspondsTo` used to describe the structure of artifacts.

However, we also see more complex patterns that are part of a technology-specific pattern catalog described in the following.

An EMF Pattern Catalog

The complex placeholder technology, used to illustrate our method, is the Eclipse Modeling Framework (EMF). This technology was part of our previous work on megamodeling and linguistic architecture (e.g., in [HHL+17]). In ongoing efforts over the years, we have developed a pattern catalog of EMF. It is published in [HHL18].

The catalog is an abstract summary of what we aim to know about EMF, i.e., some list of abstractions that we are especially interested in. We can see this as our requirements that we need to mine with our method. The patterns covered in the catalog are mined in our case study using our Datalog based method. It is a proof-of-concept and the evaluation of our proposed method.

The core of EMF (and the catalog) is a code generation process, almost similar to our running example. The EMF code generation may occur in different ways in repositories, subject to the *presence* of different types of artifacts, possibly with different *multiplicities* and in different *combinations*. In [HHL18], we published the catalog for EMF which covers the basic 'artifact' types: *Ecore Package*, as identified in '.ecore' files where one such file can possibly define several such packages (in our

Table 4.1: An *EMF* Repository Pattern Catalog

Id	Cls.	Artifacts	Description and cause
Single artifact patterns			
E	Pres.	• Ecore Pkg.	The presence of an Ecore Pkg. in '.ecore' files as root or subpackage.
J	Pres.	• Java Pkg.	The presence of a Java Pkg.
G	Pres.	• Genmodel Pkg.	The presence of a Genmodel Pkg. in '.genmodel' files as root or subpackage.
C	Pres.	• Customized Java Pkg.	The presence of a Java Pkg. with customized interface or implementation.
Double artifact patterns			
EJ1	Pot. Incomp.	• Ecore Pkg. • Java Pkg. (m[a]) [a]Missing	A Java Pkg. cannot be found for a given nsURI as extracted from some Ecore Pkg. This is only a potential incompleteness, because a Java Pkg. could be potentially derived, if no customization is intended.
EJ2	Def. Incomp.	• Ecore Pkg. (m) • Java Pkg.	An Ecore Pkg. cannot be found for a given nsURI as extracted from some Java Pkg. This is a definite incompleteness because the Java Pkg. is derived and thus, the underlying primary artifact (the Ecore Pkg.) should also be in the repository.
EJ3	Pres.	• Ecore Pkg. • Java Pkg.	The presence of a Java Pkg. and Ecore Pkg. with the same nsURI. One Ecore Pkg. can correspond to many Java Packages.
EE	Def. Incons.	• Ecore Pkg. • Ecore Pkg.	An Ecore Pkg. with at least one competing Pkg. with the same nsURI.
EJc1	Def. Incons.	• Ecore Pkg. • Java Pkg.	An Ecore classifier contained in an Ecore Pkg. with a corresponding Java Pkg., but without a corresponding Java classifier (based on name comparison). For instance, one may have forgotten to rerun the generator after adding a classifier to the model.
EJc2	Def. Incons.	• Ecore Pkg. • Java Pkg.	A Java class that is part of the Java Pkg. with a corresponding Ecore Pkg., but without a corresponding Ecore classifier (based on name comparison). One may have forgotten to remove the Java classifier by hand, after deleting it from the mode, as the generation does not delete code by default.
Triple artifact patterns			
EJJ	Pres.	• Ecore Pkg. • Java Pkg. • Java Pkg.	An Ecore Pkg. with at least two corresponding Java Packages.

running example this is the metamodel); *Java Package* – an actual Java package containing derived classes according to a metamodel, a factory, and a package description; and *Generator Package* as identified in '.genmodel' files (not present in our previous running example).

Artifacts of these types can be related in certain ways in a repository. In fact, by checking on certain relationships, e.g., by determining the *absence* of certain artifacts or elements thereof, we may infer cases of *incompleteness* or *inconsistency*, where these are either *potential* or *definite* problems of EMF usage in the repository.

Table 4.1 lists patterns organized along these different dimensions (artifact type, presence, incompleteness, inconsistency, potential versus definite). It groups them by cardinalities of artifacts: single, double, and triple artifact patterns. We exclude patterns related to XMI-based persistence in this thesis, as we have faced scalability issues in their extraction. Generally, further work is needed to arrive at a more comprehensive catalog for EMF. Detailed data on the presence of such patterns in

GitHub repositories will be reported in the case study included in this chapter.

4.4 Technical and Methodological Improvements

The previous background section introduced a blueprint of our repository mining method, illustrated for a running example. In particular, we show how to classify two artifacts, involved in a code generation scenario, to correspond to each other. We use this to produce a complex classification in terms of `correspondsTo`, classifying the relation between two artifacts. Datalog rules used for the inference are closely related to previous work on architecture recovery [MMW02, MT01, TM03], source code querying [HVdM06], and static program analysis [BS09, SBEV18].

In the following, we fill the gaps that have been omitted in the background section, that allow the fluent integration of the repository and its fragments, into the analysis with Datalog. We focus on heterogeneous content, in that we aim at accessing different artifact types, like, the file system, XML, Java and MDE artifiacts.

4.4.1 Representational Mapping

In the previous example, we have 'manually' added some facts to the reasoning. Those facts did not have incoming edges in the precedence graphs, shown until this point, like `manifestsAs`, `partOf`, `elementOf`, and `id`. Other facts, like `correspondsTo`, `deepCorrespondsTo`, `notUpToDate`, and `sumNotUpToDate`, have been derived by Datalog rules. This simplification applies to the background section. Our method provides dedicated integration mechanisms for this.

In related work, this integration is often called the *representational mapping*, or *language synopsis*. However, related work typically focuses on integrating just a single language, where we already have an AST, e.g., provided by the IDE.

In this work, we provide two integration mechanisms, doing the representational mapping. Both solve a limited version of an O/R-Mapping problem, between the revision and its fragments (where access is granted by object-oriented APIs, such as, JGit, JavaParser, JDT or DOM) and the Datalog reasoner (that works with relational data). We do not meet the full complexity of an O/R-Mapping, since for our use case, a partial mapping suffices, and the revision content typically does not change during analysis.

However, we still handle difficulties related to the identity and equality of objects, and we need a referencing mechanisms for objects to be used in the fact representation. In this section, we discuss the differences between the two integration mechanisms that we use. We described a referencing mechanism in Sec. 4.4.2.

The central difference between the two mechanisms is the following. The details will be described below.

- We may use **finite relations** to integrate revisions and its fragments. In SQL or map-reduce, we would consider this as tables or collection that we run queries on, and that we know in advance. This mechanism is essentially the same as in previous work, like on architecture recovery [MMW02, MT01, TM03], source code querying [HVdM06], and static program analysis [BS09, SBEV18].

- We may use **infinite relations** to integrate revisions and its fragments on demand. Such mechanism calls function that traverse the content of the revision and its fragments.

Finite Relations

The first integration mechanism adds facts to the data upfront to analysis. This requires a finite set of objects and references to be known in advance.

The integration mechanism requires declaring an O/R-mapping on how the content, available through an object-oriented API, can be converted into the fact representation. It can be written in any general purpose programming language adding facts over a Datalog interface.

This basic integration mechanisms may work well for scenarios where the size of the data is limited and the object-oriented structure, provided by an API, is well understood. It suffices for the related methods in architecture recovery [MMW02, MT01, TM03], source code querying [HVdM06], and static program analysis [BS09, SBEV18], as they all focus on a homogeneous scenario, with a well-understood language, and where we get the AST 'for free' (e.g., provided by the IDE).

Infinite Relations (Functions)

Often such basic representational mapping runs into limitations if the integrated object-oriented structure, provided by an API, is too big, too complicated, and when we need additional (Datalog) logic to fully understand the structure. We assume that this is a problem in a heterogeneous technology stack, e.g., as we may first understand aspects of the build system, to conclude on how to integrate ASTs.

The second mechanism to do representational mapping handles such limitations, by tightly integrating it into the Datalog reasoning. We use infinite relations that can be triggered by rules, which allows a flexible integration (see [GHLZ13] for general details on infinite relations). In the remainder, we will refer to infinite relations as 'functions'. We give this background in the following:

Consider the task of integrating an API realizing a basic arithmetic function that adds 1 to an existing number. In Datalog syntax, it is written as `Y = X + 1`. Without knowing which X is relevant to analysis, the X and Y tuples cannot be materialized as facts upfront to analysis. This problem thereby exceeds the capabilities of the first integration mechanism. However, the function still provides reasonable semantics when `X` is known (in Datalog terminology, when `X` is grounded).

The same problem appears when flexibly accessing content of a repository. We allow arbitrary functions to fully customize the access over object-oriented APIs to repository content. This includes functions that apply: parsing, querying, decomposing artifacts, computing metrics (cyclomatic complexity) and other functionality (implemented in a general purpose programming language). The crucial point is that the function must be pure, i.e., it returns the same output for the same input. This property is typically assured when using repository related APIs for querying purposes.

The following rule shows how the integration of the file and folder structure can be done by the functions `DecFs`, `IsFile`, and `IsFolder`. They are used to access and traverse files and folders over the JGit API. We highlight functions by underlining.

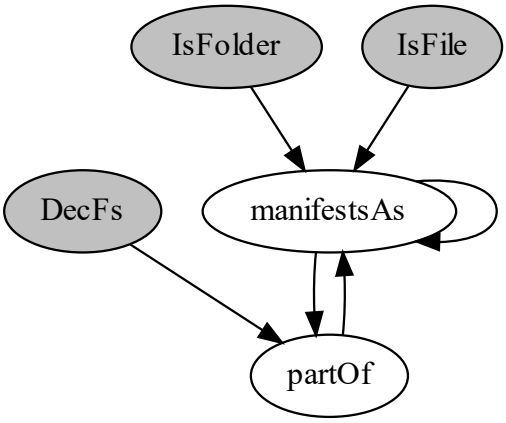


Figure 4.6: Precedence graph including repository content integration, highlighted by gray background of predicates.

```
1 // Root folder of the repository.
2 manifestsAs('repository:/', Folder).
3
4 // Recursive decomposition using an XPath query.
5 partOf(?part, ?x) :-
6     manifestsAs(?x, Folder),
7     ?part = DecFs(?x, "/*").
8
9 // Classification of folders and files.
10 manifestsAs(?p, Folder) :- partOf(?p, ?x), manifestsAs(?x, Folder), IsFolder(?p).
11 manifestsAs(?p, File) :- partOf(?p, ?x), manifestsAs(?x, Folder), IsFile(?p).
```

Listing 4.10: Decomposing a folder using infinite relation DecFs.

The integration starts with one initial fact, defining the root of the repository to be a folder (line 2, Listing 4.10). Then, the file system is successively decomposed (line 5-7) driven by the function `DecFs` (short for decompose file system), where the first argument of the function is a folder, and the second an XPath for realizing arbitrary queries to a file system starting at the folder (we have just decided to use XPath; other query languages may also apply, in general, we face a high-order function passed to Datalog). This rule produces new `partOf` facts. The functions `IsFile` and `IsFolder` check whether the new parts are files or folders; the rules (line 10 and 11) produce corresponding `manifestsAs` facts. All functions operate on file paths that refer to files and folders.

New classifications as folders assure that the decomposition is triggered recursively until no more folders are added. Accordingly, the precedence graph (Figure 4.6) contains a cycle involving predicated `partOf` and `manifestsAs` and a self reference for `manifestsAs`.

Functions plugged into the Datalog reasoner enable the same flexibility as the foreign function interface `tmapHom` or `tmap`, presented in the context of the map-

reduce solution (Chapter 3).

4.4.2 Referencing the Repository and it Fragments

As previously described, we do the integration by chaining (high-order) function calls to compose access of arbitrary object-oriented structure. The input and output of functions thereby directly refers to object-oriented structure. We need to assure that such referencing keeps intact.

Uninterpreted Function Terms In the previous examples, we have used path constants to refer to files and folders. In this section, we derive a general solution using an inductive approach:

- We assume that we have a unique identification of the repository root. All integration starts by function calls taking the root as input.
- If we have a unique identification of the input of a function, we also have an identification for its output, that is, the uninterpreted function applied to the identification of the input. As long as all functions are pure, such referencing mechanisms will always provide an identification that exactly describes the access to the same object (or objects) part of the revision.

This approach helps to chain function application without the need to define elaborate ways of mapping data between the object-oriented and the relational world. If the literal form of the reference is needed, e.g., the content of a file, the string representation of an AST, or the timestamp of a commit, a special function may return it.

The identification of files and folders by path is the same as an uninterpreted application of a selection, traversing the children of a folder. Take the following chain of function applications to the root of the repository $select(\text{'pkg'}, select(\text{'src'}, \text{'reposiory:/'}))$ which corresponds to path `reposiory:/src/pkg`.

References to Heterogeneous Content Figure 4.7 illustrates how to chain function calls to access heterogeneous repository content. The chain starts at the repository root and accesses different formats like Java and XML. This is the foundation for referencing, starting at the revision root, accessing folders, files, and finally the fragments for files of different languages. Whether all the conceivable fragments of, e.g., an AST, are materialized as facts depends on the design of the rules. In our case study, we materialize the file system completely, but we materialize file content selectively. Our method does not require the exhaustive classification of the revision and its fragments. Rules may also be turned on and off depending on which classification is needed.

Reference Optimization All other aspects of referencing are up to mechanisms for *normalization* and *optimization*. Normalization can, for instance, rewrite uninterpreted high-order function applications, including XPath queries, into a chained

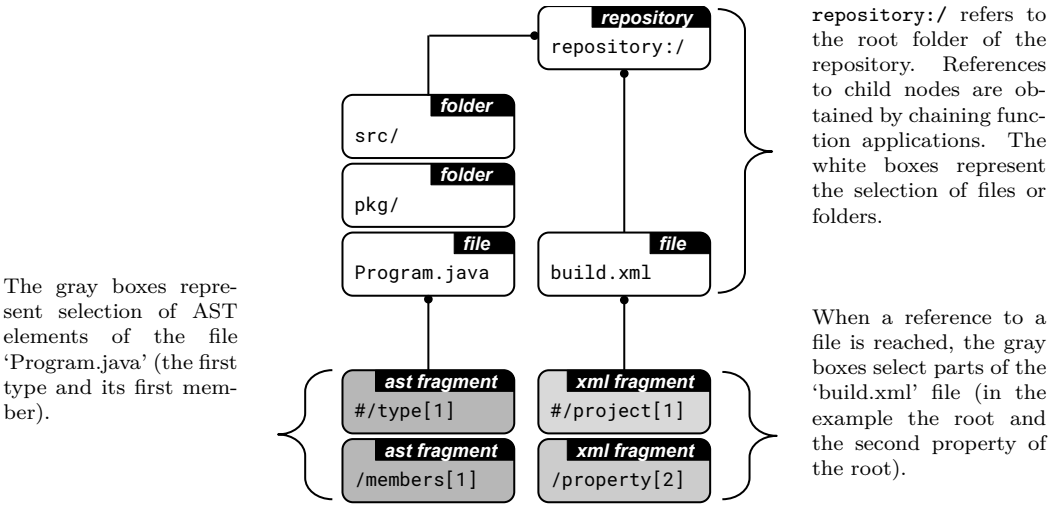


Figure 4.7: Referencing heterogeneous repository content for a small sample project containing an *ANT* and a *Java* file. The chained application of selection functions goes from top to bottom.

application of selection, traversing the direct containment relation (if there is such). This is beneficial in that different XPath queries, that point to the same files or folder, can be identified as identical after normalization.

Optimization, like memoization of intermediate results, can be used to avoid recomputing expensive parsing results during the execution of the Datalog program.

Flat Collection Output Functions returning collection types are essential to our method, e.g., for decomposing Folders or ASTs into its fragments. A selection applied to a collection can be used to disambiguate between the collection and the individual members. For instance, function `?part = DecFs(?x, "/*")` binds *?part* to individual members of the decomposition and not to the collection. This decision is up to the implementation of the function. It is somehow analogue to the difference between `map` and `flatMap` in classical map-reduce.

4.4.3 A Catalog of Accessor Functions

We have realized the previous aspects on accessing repository content, providing a prototypical list of high-order accessor functions. Most of the functions can be parameterized by XPath to customize the access. The list can be used to integrate standardized formats and structures part of repositories, such as the files, folders, Java AST and XML AST. The functions are also used in the case study and are shown in Table 4.2.

Table 4.2: Primitives for accessing folder structure (in general) and file content for *XML* and *Java* (in the case study).

Primitive	**Parameters and Description**
`IsFile`	`(artifact)` Matches if the `artifact` can be accessed as a file.
`IsFolder`	`(artifact)` Matches if the `artifact` can be accessed as a folder.
`Extension`	`(file)` Returns the extension of a `file`.
`NameFs`	`(artifact)` returns the name of a file or folder.
`DecFs`	`(folder, xpath)` Returns references to files of folders by applying an XPath expression `xpath` on the repository starting from the given `folder`; returns the individual (flat) results.
`XmlWellformed`	`(file)` Parses the content of a `file` and matches if the content is well-formed *XML*.
`DecJava`, `DecXml`	Variations on `DecFs` working on *Java* ASTs or *XML* trees as opposed to the file system. XPath also applies to the output of object-oriented parsing APIs.
`StrXml`, `StrJava`	Variations on the decomposition primitives returning the literal form of the result (and not the reference form).

4.5 Evaluation

We evaluate the method in terms of implementing a prototype and running a complex case study. We use the prototype to recover the abstract patterns presented in the pattern catalog (Sec. 4.3.2) on GitHub.

The evaluation will proceed as follows: We first located repositories with traces of EMF usage. In a first stage, we produce a foundational understanding of the repositories and the build systems. We limit following up more complex classifications to well-understood EMF project layouts, for which we assumed insights to be more comprehensible. We then build more advanced analysis of MDE related artifacts. Eventually, we detect EMF repository patterns.

We discuss some modular Datalog rules, showing some challenges of processing XML ASTs. We will discuss the mined patterns and the mining performance.

4.5.1 Locating Repositories

We used the *GitHub* search API to locate all recently indexed *Ecore*, *Generator Model* and *Java Model* files on *GitHub* as an indication of *EMF* usage in repositories. The corresponding queries are listed in Table 4.3. A list of 5759 *GitHub* repositories was extracted from the query results[4].

Table 4.3: Queries for locating repositories through *GitHub* API.

Evidence	**Query**	**Extension**
Java Model	"extends EObject {"	java
Ecore Model	EClass	ecore
Generator Model	GenModel	genmodel

4.5.2 Initial Classification of Files by Language

We start classifying files by languages, as this is a prerequisite for diving deeper into the content of a repository, e.g., at the level of parse trees or ASTs. The following rules classify files by a language, adding `elementOf` facts for files and the language at hand (Listing 4.11).

```
1 elementOf(?x, Java) :- manifestsAs(?x, File), "java" = Extension(?x).
2 elementOf(?x, XML) :- manifestsAs(?x, File), XmlWellformed(?x).
3 elementOf(?x, Gradle) :- manifestsAs(?x, File), "build.gradle" = NameFs(?x).
4 elementOf(?x, Ecore) :-
5     manifestsAs(?x, File), elementOf(?x, XML), "ecore" = Extension(?x).
6 elementOf(?x, Manifest) :-
7     manifestsAs(?x, File), "MANIFEST.MF" = NameFs(?x),
8     partOf(?x,?folder), "META-INF" = NameFs(?folder).
9 elementOf(?x, Ant) :-
```

[4]For what it matters, at the date the search was applied, the API search limit was circumvented by recursive query segmentation which splits a query by setting an upper and lower bound in file size based on the returned number of total results. This process may miss some results. The search API only considers heads of default branches and files smaller than 384 KB.

```
10    manifestsAs(?x, File), elementOf(?x, XML), "build.xml" = NameFs(?x).
11 elementOf(?x, Pom) :-
12    manifestsAs(?x, File), elementOf(?x, XML), "pom.xml" = NameFs(?x).
```

Listing 4.11: Rules for basic language classification.

We classify Java, Ecore and XML files, as those three kinds are the most important languages in the context of EMF. XML is not classified by extension, but by well-formedness, as it often appears with domain-specific file extensions in EMF. We also classify files related to the used build systems.

4.5.3 Selection of Repositories

The rules of Listing 4.11 are the starting point to recover the 'repository layout' in terms of usage of build systems, project dependencies, and other aspects. We developed the following classifiers that apply to the repository as a whole as an extension to the EMF pattern catalog:

Homogeneous versus **heterogeneous build system:** We search for traces of Manifest, POM, Gradle, and ANT, as modeled by the rules in Listing 4.11. In the homogeneous case, only one such technology is used; otherwise, we apply the heterogeneous classifier. We assume that the heterogeneous situation is harder to understand in terms of project dependencies.

Single component versus **multiple components:** Based on an analysis of project dependencies, as described in more detail below, we determine the number of components. We assume that repositories with multiple components are special. Such a repository may be, for example, a 'zoo' [KSW+13]. Note that a single component can still imply the presence of multiple (dependent) projects, part of the repository.

Variants: This classifier applies when we locate different versions of the same project in a repository based on the analysis of project dependencies. We assume again that repositories with variants are special. Such a repository may capture, for example, versions in a migration process.

EMF's default is the use of Manifest files for defining OSGi projects and dependencies. We decided to only include homogeneous repositories using Manifest files in our case study. The heterogeneous build system of repositories calls for future work. The analysis of dependencies is based on 'Bundle-SymbolicName' elements in Manifest files. Listing 4.12 presents the rules for inferring the occurrence of declarations (predicate `decOccures`), references (predicate `refOccures`) and OSGi dependencies between Manifest files (predicate `dependsOn`):

```
1 // Extraction of Bundle-SymbolicName declaration.
2 decOccurs(?file, ?declaration) :-
3    elementOf(?file, Manifest),
4    ?x = StrManifest(?file, "Bundle-SymbolicName"), // Extract a manifest property.
5    ?declaration = ReplaceAll(?x, "(;[^,]*)|\\s", ""). // Replace details.
6
7 // Extraction of Bundle-SymbolicName references.
8 refOccurs(?file, ?reference) :-
```

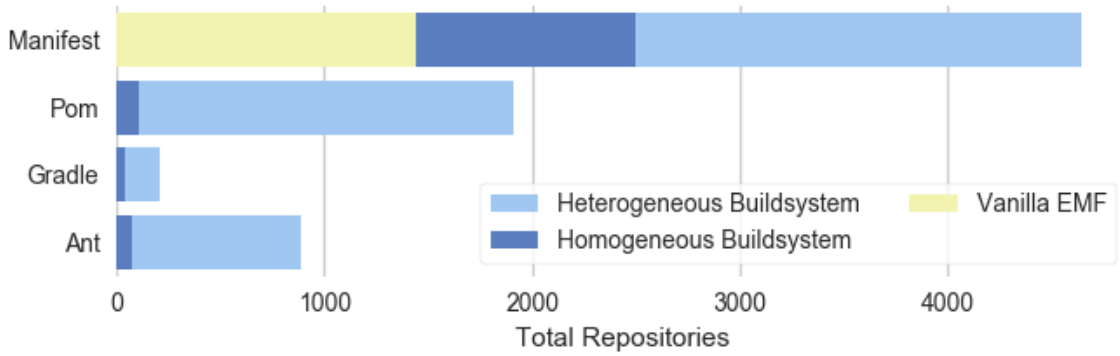


Figure 4.8: Number of repositories with a particular build system, further partitioned into homogeneous versus heterogeneous case.

```
9     elementOf(?file, Manifest),
10    ?x = StrManifest(?file, "Require-Bundle"), // Extract a manifest property.
11    ?xi = ReplaceAll(?x, '("[^"]*)"', ""), // Replace details.
12    ?reference = Split(?xi,','). // Split multiple dependencies.
13
14 // Creating dependency fact.
15 dependsOn(?deca, ?decb) :-
16    elementOf(?a, Manifest), elementOf(?b, Manifest),
17    decOccurs(?a, ?deca), refOccurs(?a, ?decb), decOccurs(?b, ?decb).
```

Listing 4.12: Rules for extracting OSGi declarations, references and dependencies.

The function `StrManifest(file, property)` is a specialized decomposition function of a Manifest file, returning a Manifest property in literal and not in reference form. In Listing 4.12, `?x` is assigned to the 'required' or 'defined' bundles. The chains of function applications on `?x`, invoking `ReplaceAll` and `Split`, continues processing `?x` in literal form according to the standard String processing semantics.

We exclude repositories with duplicated declarations, where `decOccurs` refers to the same declaration from different files (classifier *Variants*). This can, for instance, be checked by aggregation. We apply an algorithm for detecting connected components to the `dependsOn` relation. We exclude repositories with multiple components.

The results of the selection steps are depicted in Fig. 4.8. In what follows, we only consider repositories with a single component, Manifest usage only, and no variants. We refer to these repositories as 'Vanilla *EMF* repositories'. There are 1437 such projects. These are the projects considered for mining.

4.5.4 Mining the EMF Pattern Catalog

For brevity, we focus on the detection of correspondence between Java and Ecore models, as introduced in the motivation section. In particular, we show the decomposition of an Ecore model which is persisted as XML; we omit the rules for detecting and decomposing the Java package; we also omit handling Ecore sub-packages.

Consider the beginning of a small Ecore sample file as follows:

```
1 <ecore:EPackage ... name="fsml" nsURI="http://www.softlang.org/metalib/emf/Fsml" nsPrefix
      ="fsml">
2    <eClassifiers xsi:type="ecore:EClass" name="FSM">
```

```
3          ...
```

Listing 4.13: The first lines of a sample Ecore file.

The following rules decompose the Ecore model into root package and nested classifiers:

```
1  // Decomposition of the Ecore file into ...
2  // ... the root package.
3  partOf(?part, ?x), elementOf(?part, EcorePackageXMI) :-
4      elementOf(?x, Ecore),
5      ?part = DecXml(?x, "/ecore:EPackage").
6
7  // ... the nested classifiers in a package.
8  partOf(?part, ?x), elementOf(?part, EcoreClassifierXMI) :-
9      elementOf(?x, EcorePackageXMI),
10     ?part = DecXml(?x, "/eClassifiers").
11
12 // Extracting nsURI and name, necessary for detecting correspondence.
13 id(?x,?id) :-
14     elementOf(?x, EcorePackageXMI),
15     ?id = StrXml(?x, "/@nsURI"). // NsUri for a package as id.
16
17 id(?x,?id) :-
18     elementOf(?x, EcoreClassifierXMI),
19     ?id = StrXml(?classifier, "/@name"). // Get the classifier's name as string.
```

Listing 4.14: Decomposing Ecore into classifiers appending a nsURI.

A `partOf` relationship is inserted along the nesting structure and fragments are classified by `EcorePackageXMI` and `EcoreClassifierXMI` respectively. This decomposition is handled by the first two rules (line 3-5 and 8-10, Listing 4.14), using the function `DecXml` to construct references to the XML AST. The last two rules (line 13-15 and 17-19, Listing 4.14) extract the attributes 'nsURI' and 'name' in literal form, using `StrXml` applied to the AST. This is mandatory because the literal values 'FSM' (name) and '`http://www.softlang.org/metalib/emf/Fsml`' (nsURI) are needed rather than the references to the AST.

The rules for establishing the correspondence are similar to those for the running example, presented in the background section (with small modifications, exchanging classifier names). In total, we mine 10 patterns that arise from combinations of the heterogeneous artifact types.

4.5.5 Results

> The empirical results on the EMF pattern catalog, and the catalog, are associated with Marcel Heinz. We summarize the content here to be self-contained. Parts of the presentation of the results has been improved in this thesis. However, we only show the results because of their relevance to a credible case study, that serves us as a prove-of-concept for the method we contribute in this chapter.

This chapter focuses on the technical aspect of declarative mining of a heterogeneous technology stack. The previous section proves that this is possible in this

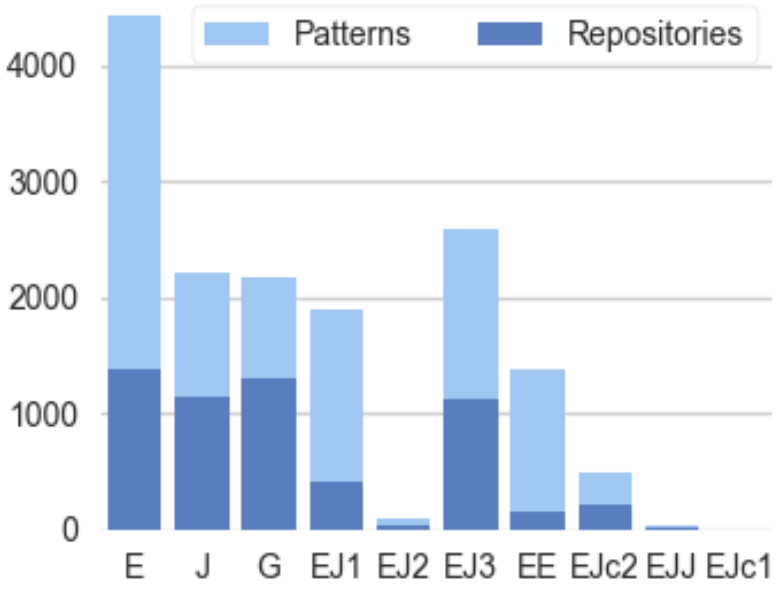


Figure 4.9: Overall pattern sum and the number of repositories a pattern occurs in.

concrete case (prove-of-concept). For completeness, we also discuss the results of our mining.

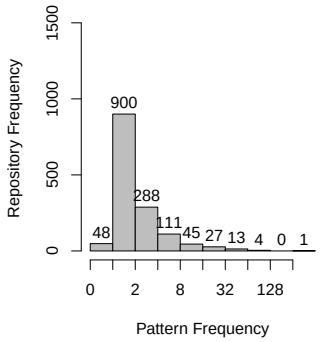
The results of the case study, applied to 1437 Vanilla EMF repositories, are summarized in this section. In the following, we will: i) present raw data on the individual patterns, ii) model group membership (definite incompleteness) as a function of *repository popularity* (stars) and iii) model how the mining performance relates to the number of analyzed files in a repository.

Raw Numbers of Patterns

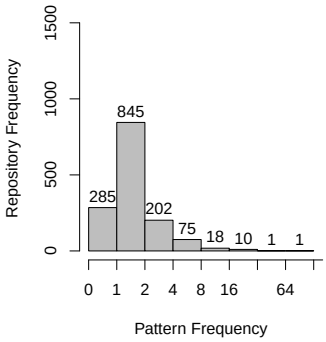
Fig. 4.9 gives a first overview on the number of pattern occurrences. It shows two overlapping bars that represent: i) the overall sum of patterns across all repositories (light blue) and ii) the number of repositories that contain the pattern at least once (dark blue). The plot gives a first intuition on which patterns are less frequent, i.e., EJ2, EJc2 and EJJ. Pattern EE is frequent, but just in a few repositories. We did not find EJc1.

A detailed picture of the distribution of pattern frequencies over all repositories is reported in Figure 4.10. The series of plots show how often 0, 1, 2, 4, 8 ... occurrences of the same pattern are part of a repository (0 indicates absence). This view is more applicable than examining the mean and standard deviation, as the distributions are far from being normal. We also add the description of the pattern as a recap in the caption. The following observations can be made:

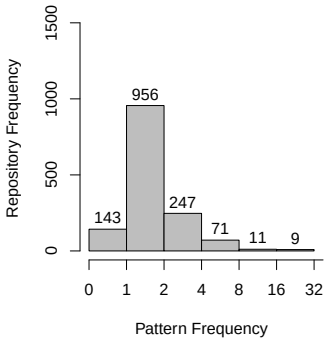
- **E** (Figure 4.10a): *The presence of an Ecore Pkg. in '.ecore' files as root or subpackage.* In almost all repositories, we find at least one Ecore package. Only 48 of the 1437 Vanilla EMF repositories miss it. Such absence might be caused by some additional process deriving the Ecore model. In most cases (900), we exactly face one package. However, there are also repositories with more than a single package. One repository contains over 256 Ecore packages.



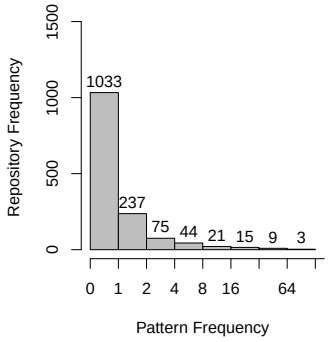
(a) **Pattern E**: The presence of an Ecore Pkg. in '.ecore' files as root or subpackage.



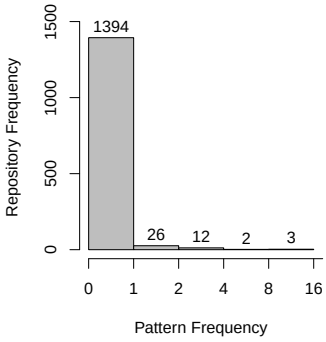
(b) **Pattern J**: The presence of a Java Pkg.



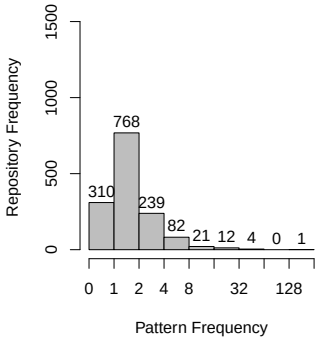
(c) **Pattern G**: The presence of a Genmodel Pkg. in '.genmodel' files as root or subpackage.



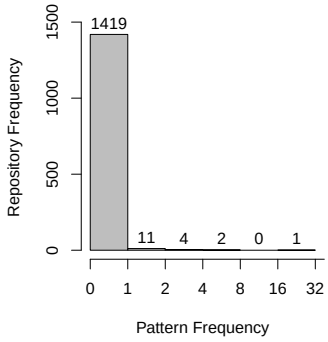
(d) **Pattern EJ1**: A Java Pkg. cannot be found for a given nsURI as extracted from some Ecore Pkg.



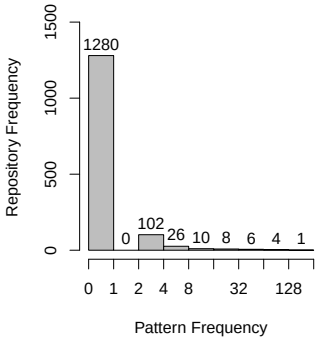
(e) **Pattern EJ2**: An Ecore Pkg. cannot be found for a given nsURI as extracted from some Java Pkg.



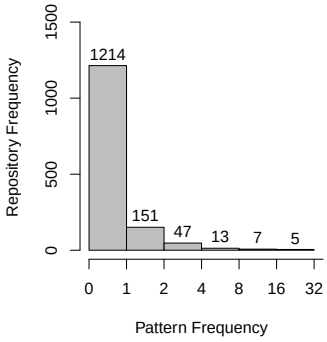
(f) **Pattern EJ3**: The presence of a Java Pkg. and Ecore Pkg. with the same nsURI. One Ecore Pkg. can correspond to many Java Packages.



(g) **Pattern EJJ**: An Ecore Pkg. with at least two corresponding Java Packages.



(h) **Pattern EE**: An Ecore Pkg. with at least one competing Pkg. with the same nsURI.



(i) **Pattern EJc2**: A Java class that is part of the Java Pkg. with a corresponding Ecore Pkg., but without a corresponding Ecore classifier (based on name comparison).

Figure 4.10: The distribution of patterns over Vanilla EMF repositories.

- **J** (Figure 4.10b): *The presence of a Java Pkg.* In comparison to the previous pattern, the number of repositories without Ecore Java packages is high with

285. This is reasonable, since often generated code is not committed to the repository.

- **G** (Figure 4.10c): *The presence of a Genmodel Pkg. in '.genmodel' files as root or subpackage.* The number of repositories without a generator package is somewhere in between the previous two cases, E and J, with 143. Often, the generator model can be derived, but there are also points where customization might be necessary, which requires committing the generator.

- **EJ1** (Figure 4.10d): *A Java Pkg. cannot be found for a given nsURI as extracted from some Ecore Pkg.* Having no corresponding Java package for an Ecore model can happen if the generated code is not uploaded (potential incompleteness). In 1033 repositories, we do not face the pattern. However, in 404 repositories we face this pattern at least once, indicating that the repositories definitely do not customize code and potentially rely on a proper generation process to run after checkout.

- **EJ2** (Figure 4.10e): *An Ecore Pkg. cannot be found for a given nsURI as extracted from some Java Pkg.* This is the first pattern that indicates a definite incompleteness, since we know that if code is uploaded, also the model needs to be uploaded for the shake of consistency. The distribution of such pattern looks different to the previous patterns. In only 43 of the Vanilla repositories (3%) this pattern is faced at least once. Some sort of feedback to developers that introduce such problem might resolve such situation. EMF does not inform uses on this.

- **EJ3** (Figure 4.10f): *The presence of a Java Pkg. and Ecore Pkg. with the same nsURI. One Ecore Pkg. can correspond to many Java Packages.* This is the regular correspondence present in many repositories. However, there is still an amount of 310 repositories that misses such correspondence (possibly because they also lack Java code, i.e., pattern J, 258 repositories).

- **EJJ** (Figure 4.10h): *An Ecore Pkg. with at least two corresponding Java Packages.* This is a corner case, employing a rare feature of EMF to derive potentially different implementations of models. We only face it in 18 repositories.

- **EE** (Figure 4.10h): *An Ecore Pkg. with at least one competing Pkg. with the same nsURI.* We face this pattern in 157 repositories at least once. Duplication is not common practice and classified as definitely inconsistent. However, it still seems to be a common habit, potentially involved in situations of prototyping EMF solutions.

- **EJc1** (not matches): *An Ecore classifier contained in an Ecore Pkg. with a corresponding Java Pkg., but without a corresponding Java classifier (based on name comparison).* In other words, we miss a Java class. We do not find any occurrence of this pattern which indicates that, if the Java code is uploaded, nobody misses rerunning the generation after adding a classifier.

- **EJc2** (Figure 4.10i): *A Java class that is part of the Java Pkg. with a corresponding Ecore Pkg., but without a corresponding Ecore classifier (based on*

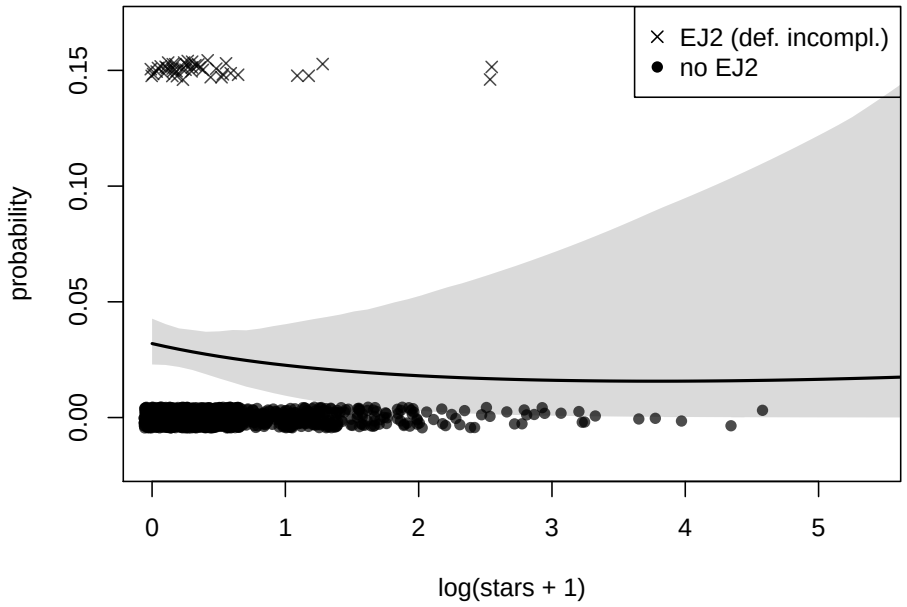


Figure 4.11: Probability of facing EJ2 computed by a logistic regression

name comparison). In other words, we find a useless Java class in the generated code. We find this pattern often with 223 repositories that have at least one occurrence. Such a pattern is not inconvenient, as the generator process does not automatically delete code (to protect against the deletion of handwritten code). We face a definite inconsistency.

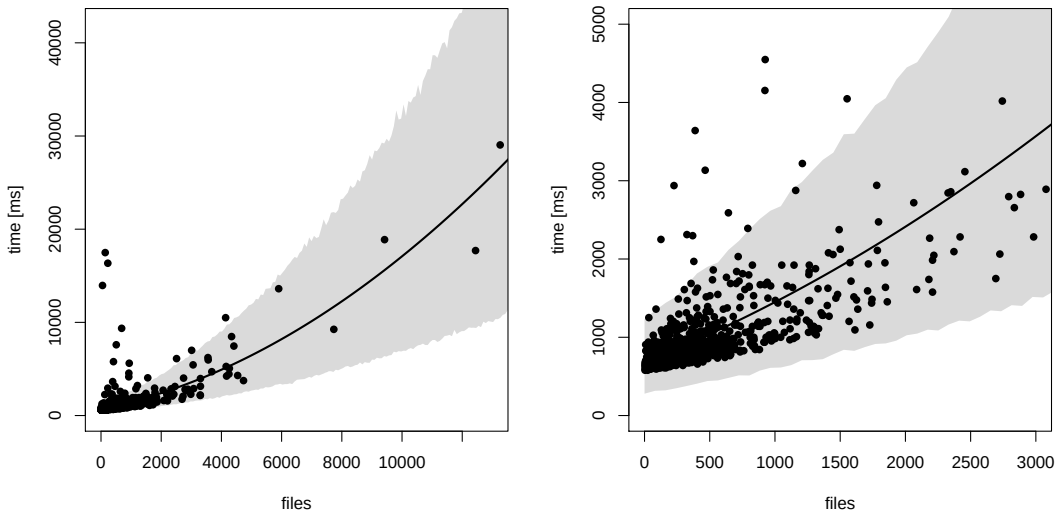
4.5.6 Modeling Group Membership

In this section, we model the existence of problematic pattern, as a function of repository popularity. We assume that problematic patterns are less frequent in popular repositories. We focus on definite incompleteness in terms of pattern EJ2, which is the most problematic in our catalog.

We model the presence of EJ2 as a function of the repository popularity in terms of stars. The stars are stated-log transformed, since popularity often shares an exponential growth. We implement a Bayesian logistic regression (with flat priors).

The resulting model is depicted in Figure 4.11, showing the probability of a repository including an EJ2 pattern, over the range of possible popularity values. The line depicts the mean probability. The shaded area shows the Bayesian credibility interval (95%) around the mean. We plot all repositories as dots or crosses (depending on the presence of EJ2), with jitter on the x and y-coordinates at y-coordinate 0.0 or 0.15.

The pot shows a decreasing trend of EJ2 membership with popularity. The 95% credibility interval for the slope, however, may include positive trends (the interval lies between -1.27 and 0.26). For a star count of 0, the credibility interval for the probability of facing EJ2 ranges from 2.2% to 4.2%.



(a) All repositories (b) Low file counts

Figure 4.12: Performance of the Mining

4.5.7 Analyzing Mining Performance

We analyze the performance of our prototype by modeling the mining time as a function of repository files. We use a Bayesian model (with flat priors).

The output variable mining time is modeled following a gamma distribution (typical for distance and duration since it is constrained to be positive and continuous with a peak above 0). We use a parameterization setting the distribution shape as a model parameter, and the distribution mean as a polynomial function of time.

The resulting fit is depicted in Figure 4.12, where the left plot show all repositories (files vs. mining time) and the right is limited to low file counts. We plot the mean of the distribution as a line over file counts and the 95% credibility interval of expected mining times as a shaded area. The plots show some weaknesses of the model for low file counts. We can see this in terms of the interval that does not align with the actual lower border of the observations (see the lower part of Figure 4.12b). We assume that this model is still sufficient to make statements on the mining performance.

We aim to examine the linear $O(n)$ and quadratic effort $O(n^2)$ of running our method on files. The plots hint at an evident quadratic trend. The model reports that the average startup time lies between 0.65 and 0.7 seconds; the time needed to process a single file lies between 0.57 and 0.77 milliseconds. Further, we have a factor between $5.47E-05$ and $14.29E-05$ for the quadratic effort, increased by each file.

4.6 Conclusion

In this chapter, we demonstrate how to use declarative logic programming, in particular Datalog, to mine a repository with a heterogeneous technology stack. We focus on mining EMF usage patterns. Such patterns consist of a heterogeneous mixture of XML, Java and specific MDE artifacts. We also discuss the results.

We propose a method to improve the definition of functions that abstract over a repository with a heterogeneous technology stack, by using concepts from declarative logic programming and combining them with ideas on megamodeling and linguistic architecture. We reproduce existing ideas on declarative logic programming with languages close to Datalog, coming from architecture recovery, source code querying, and static program analysis, and transfer them from the analysis of a homogeneous to a heterogeneous technology stack. We provide a prove-of-concept of such method in a case study.

In the next chapter, we will start to discuss methodological challenges of working with repository data, leaving the rather technical challenges of the computation behind. Choosing the appropriate methodology for answering real empirical questions based on raw abstractions/variables mined from repositories and its fragments, is highly challenging. The previous parts, included in Chapter 3 and 4, can be seen as the technical tools operating in the background. Such technical solutions are one important aspect of a good methodology in MSR/ESE to collect data.

Chapter 5

Simulation-Based Testing

5.1 Introduction

The reviewing process of empirical work is challenging because quality criteria on the validity of methods and results are hard to define and communicate. For example, the *International Conference on Software Maintenance and Evolution* (ICSME) includes a dedicated category for empirical work. Calls-for-papers in the years 2021, 2022, and 2023 include this statement to describe the reviewing of such a category:

An empirical work is a *'[...] paper in which the main contribution is the empirical study of a software evolution technology or phenomenon. [...]* ***The authors should provide convincing arguments [...] why certain methods or models are needed. Such a contribution will be judged on its study design, the appropriateness and correctness of its analysis, and its discussion of threats to validity.*** *Replications are welcome.'* (copy from the call-for-papers of ICSME 2021, 2022, and 2023)

Similar statements on the validity (or correctness) of work can be found in other empirical fields. While reproducibility and replicability are somewhat understood [CSFG19, SSM18], standardized and operational ways to define and communicate the validity of methods and results, and the threats to it, are less understood. An example of recent work that points out challenges of defining and communicating threats, in the context of program comprehension experiments, is [BWGW23]. This thesis focuses on this latter aspect.

5.1.1 Meta Research Questions

In this thesis, we try to better understand the general problems related to the validity of methods and results in empirical research with a focus on scenarios coming from the field of Mining Software Repositories (MSR) and Empirical Software Engineering (ESE). We hope that this effort also contributes to an improved understanding in other empirical fields.

We derive the following meta research questions, that we later instantiate for concrete empirical studies:

- **RQ 1:** What assumptions of ESE and MSR studies can be operationalized?
- **RQ 2:** What is the impact of such assumptions on the study results?

The first question asks for a more operational way of expressing assumptions about an empirical scenario. Such operational form can be helpful in the communication and the reviewing process. This is the foundation to discuss validity. The second question asks about the impact of assumptions on the results of a study. This is relative to the method used to produce results and alternatives to it.

5.1.2 Relevance

Our research questions matter from an author and a reviewer perspective: Authors of empirical research try to assess and explain the validity of their empirical research. Reviewers need guidelines on what validation they should search for in submissions.

5.1.3 A New Validation Strategy: Simulation-based Testing

This thesis presents a new strategy that operationalizes statements about the validity of empirical studies. In this chapter, we use the term *strategy* as an alias for a *high-order method*.

We operationalize assumptions on an empirical research scenario, typically informal in a paper, by simulations that produce artificial data and results. The impact of a used method on its results can be checked in such a transparent setting.

We encourage researchers to **submit simulation code as validation artifacts in the reviewing process of empirical work** to define and communicate properties related to the validity of their methods.

5.1.4 Meta-Validation

What we propose is a general strategy (or method) to validate methods and results for a concrete empirical scenario. To provide a validation on our part, we applied our general method to 6 real scenarios examined in published studies in MSR and ESE. In each case, we instantiate our meta research questions and show how we can answer them.

We prove that benefit can be expected by such additional validation artifacts. We show that we can either: i) support validity ii), threaten validity, or iii) prove invalidity of the used methods and results by simulated scenarios.

5.1.5 Summary of this Chapter's Contributions

- We present simulation-based testing for MSR/ESE studies to operationalize threats to validity.
- We evaluate simulation-based testing, showing the relevance of simulation-based testing by applying it to published studies in MSR/ESE.
- We provide an initial catalog of relevant threats to MSR/ESE studies using simulation-based testing.

The full simulation code covered in the remainder of this chapter **can be found online**[1] and may be used for reproducing the presented tests. The code is written in R but can be translated into other languages.

5.1.6 Summary of the Delta to the Publication

This chapter keeps close to the previous publication [HL22] and to a more recent journal version [HL23]. We adjust some minor structural aspects, to better fit to the overall metamodel of this thesis.

5.1.7 Micro road-map of this Chapter

This chapter follows a limited version of the metamodel defined in the introduction (Sec. 1.6). In particular, Sec. 5.2 begins with a motivation; the background section is skipped; Sec. 5.3 introduces simulation-based testing using a basic example; Sec. 5.4 evaluates simulation-based testing by applying it to existing studies in MSR and ESE.

[1] `https://github.com/topleet/MSR2022`

5.2 Motivation

We start with the discussion of existing strategies for the validation of empirical work that come close to our general strategy. We add some known examples of empirical work where validation has gone wrong. This motivates the complexity of validation.

The section can be skipped if a positioning in the context of reviewing and validating empirical work is not of immediate interest upon first reading.

5.2.1 What is a Valid Method?

Reasoning about the validity of a method used in empirical research is hard:

- There might be trivial problems. An example of a study, where columns in the data are accidentally flipped, is described in [Mil06]. The authors needed to withdraw five publications because of this mistake in their method. Such trivial bugs might be a detail that nobody notices for years, but a detail that changes the results dramatically. Nothing prevents researchers from running into such problems.
- There might be more subtle problems. An instance is a study about cultures with moralizing gods in the field of anthropology, criticized in [BAB+21]. Here, a small difference in the method, in particular, on how to handle missing values in historical records, leads to dramatic changes in the results. There is no real solution to the problem, as discussed in [McE20] (page 512). Such reasoning relates very plausible assumptions on the empirical scenarios, in particular, about the origin of missing data, to the method and its incapability of producing valid results.

What can be noticed in such discussions is that it is hard to judge a method by its results. A comparison of methods might indicate differences in the results, but differences alone cannot always tell something about validity. An understanding of the empirical scenario is necessary for claims on validity.

5.2.2 Typical Strategies

Typical strategies that may help with the validity of empirical work will be listed next. The use of simulations is not established. The following list summarizes our experience with validation strategies we spotted in past publications of MSR/ESE.

- **Intuition**: Research in MSR/ESE is typically conducted, reviewed and read by software engineers. This implies that all results and empirical scenarios can, to some extent, be judged by our intuition. We often see sections in publications that trigger this, giving small anecdotes, and explaining why particular results are intuitive or not. Some examples can be found here [GS10, GdCZ19, Vok04, PCGA08, SMS16, JMF14, BFS+18]. The most characteristic text passage we may find in papers is '*... results confirms our intuition that ...*'. However, such judgment of results might be dangerous.

- **Authority**: In some cases, authors may establish authority, which creates an impression of validity. This may be done by an extensive discussion of related work or referring to previous efforts done by the authors.

- **Cookbook Methods**: In several cases, claims on the validity of methods (and parts of it) are outsourced by using previously established methods. Papers list references to previous work, using the method too, to prove validity. Established methods that have been used for a while, are for instance, the events-per-variable (EPV), introduced in [PCK+96] and used as part of MSR/ESE methods in [GdCZ19, JTH21, TH18, TMHM17, PFD11], or AIC, introduced in [Aka98], and used in MSR/ESE method like [TDH14, PFD11, RPD12, CDO+15, IYNH19]. However, if a methods works out-of-the-box in a new empirical scenario is not always clear.

- **Comparison Results**: Examining the consistency of results with previous work is also typical. This can be done by replication of previous studies, or by a meta-analysis. Typically, studies are applied in a closely similar scenario and on fresh data. An example is presented in [Moc10], where a detailed table makes explicit to which previous studies the new results conform or not. Consistency is assumed to be a good sign in favor of a study. Less formal replication, comparison, and confirmation of previous results, located in the text, rather than in a table, can be found in [CDO+15, KL17, TBP+17]. However, whether consistency alone may be taken as a sign for a valid study can be doubted. It may also be caused by the repeated application of an invalid method.

- **Comparison Methods**: The comparison of methods may indicate a difference in the results. In specific cases, as in the case of a model comparison (see the next item), the difference is meaningful. However, in general, such a difference may not necessarily indicate which method is valid. Without a clear understanding of the relation between the empirical scenario, method and results, statements about validity are limited.

- **Model Comparison Method**: We consider the comparison of models that are proxies for different hypotheses as one particular method. It is maybe the most established and useful in empirical research. Here, a comparison selects between different models by preferring those that fit the collected data best. Relevant part of such method is the protection against over and underfitting by cross-validation (used in [BSHA20, CLP+15, DBG+15, HPMY13, NFK+18]), information criteria (used in [TDH14, PFD11, RPD12, CDO+15, IYNH19]) or regularization (used in [NPK13]). Model comparison is well understood, but still limited. Model comparisons can, for instance, be misleading when talking about causality. Non-causal models are often preferred because they improve the fit. In general, associating models and hypotheses is not trivial and might go wrong.

- **Simulations**: The use of simulation is a recent trend that is currently starting to spread in teaching statistics [GHV20, McE20]. According to our knowledge, it is not a formal requirement in the reviewing process.

In this paper, we will go this new way of simulating empirical scenarios and results to support or threaten the validity of methods used in MSR and ESE.

5.3 Technical and Methodological Improvements

This section presents the original contribution.

5.3.1 A New Validation Strategy: Simulation-Based Testing

This section introduces simulation-based testing to operationalize statements about the validity of empirical research in MSR and ESE.

> **We call our strategy 'simulation-based testing' because of the analogy between writing simulation artifacts and writing test cases.**

A Simplified Empirical Study

For a structured discussion, we decompose empirical research studies into i) an empirical scenario, ii) a method, and iii) the results. This is a strong simplification.

- The *empirical scenario* is the domain-specific and more 'informal' part of a study that is the subject to research.
- The *method* consists of the steps to produce results.
- The *results* are non-trivial statements about the empirical scenario.

The following logic statement (that should not be read with formal ambitions) illustrates the argumentation on validity in many studies: *'Valid assumptions about the empirical scenario and a valid method imply valid results.'*

$$\text{empirical scenario} \wedge \text{method} \rightarrow \text{results}$$

Judging the validity of the results is often complicated and influenced by expectations. Instead, researchers judge the assumptions about the empirical scenario and the validity of the used method. Correct results are implied ($\rightarrow$). However, such argumentation requires a clear understanding of the relationship between the three parts: How do the assumptions on the empirical scenario and the method influence the results?

In this paper, we will show how to operationalize statements about the validity in terms of the relationship between the assumption about the empirical scenario, the method, and the results.

Variables and Relationships

In one form or another, an empirical study suggests a description of a data generation process:

- **Variables** label relevant data for the research scenario. Typical variables in MSR/ESE are the occurrence of defects, lines-of-code, or effect strengths. The measurement of such variables matters: Some variables can be *observed* (e.g., lines-of-code); some variables can only be observed with uncertainty (e.g.,

defects due to inaccuracy of SZZ [SZZ05]); other variables are conceptual and thereby in principle *unobserved*. They can be inferred as the result of an empirical study (e.g., effect strength of lines-of-code influencing defects).

- **Relationships between variables** describe how variables relate to each other. Relationships may be functional or stochastic. The latter invokes a notion of uncertainty. Uncertainty is handy, since we do not find exact relationships between variables. The direction of such relationships is relevant for claims on causation and to describe a process. Temporal precedence of variables sometimes limits the plausible directions in which a relationship may operate. Relationships can never be measured directly.

Different algorithms can now use relationships to *infer*, *predict*, *identify*, *learn* or *simulate* variables using other variables.

Depending on which variables are observed or not, how unobserved variables are treated, and which relationships are used, the names for the procedure and the involved algorithms may differ:

- Algorithms may *predict* unobserved defects.
- Algorithms may *infer*, *learn* or *identify* unobserved parameters relevant to a relationship. We stick to the term *identify*.
- Algorithms may *simulate* complete data sets following plausible assumptions on unobserved variables, like on the relationship between defects and lines-of-code.

The Baseline Empirical Method

We now describe a common underlying method used in many empirical studies. It is not part of our new strategy, but relevant to it. We describe it here to make this discussion self-contained.

Studies often execute a variation of the following steps to arrive at the results:

- A study's method uses the toolbox of variables and relationships to decode hypotheses about the empirical scenario as 'models'. There are often stereotypical ways of connecting variables by relationships, with well-known names and algorithmic support. For instance, there are linear (regression) models, logistic (regression) models, mixed-effect models, autoregressive models, or generalized additive models. These names just refer to blueprints that still need to be customized.
- The data collection mines a sample, for instance repositories, to replace some unobserved with observed variables.
- Algorithms attempt to identify the remaining unobserved variables in the models.
- The identified unobserved variables, often called the parameters, and the model that fits the data best is considered as the result of a study. Hypothesis are discussed respectively.

We can find comparable practice in the following papers [KSA+13, FBF+20, VPR+15, BHV16, FLHV22, SHL+19, MW00, YXF+20, JTH21, NZZ+10, RRC16, ZPZ07, ZN08, TMHI16, TDH14, TH18].

Simulation-Based Testing in a Nutshell

Judging the validity of such a method and variations of it is complicated because it is connected to assumptions on the empirical scenario. **To make the assumptions transparent, we operationalize them in terms of a simulation.**

A simulation-based test replaces the real empirical scenario, in essence the collected data, by a 'simulation' that reflects our assumptions about the empirical scenario. The simulation produces artificial data.

The simulation-based test often works in an 'opposite way' when compared to the original method. Instead of using the observed data to produce results, it reverts the algorithms, and produces artificial data for given results. The results can be purely fictional (or counterfactual) and are typically invented for the purpose of simulation.

An empirical study's original method, or any other method that potentially applies, can be run on the simulated data.

The simulated empirical scenario, the method, and the results are now transparent. The relationship can operationally be judged. The impact of our assumption on the results can operationally be judged.

> **Possible Statements on Validity:** Simulated scenarios operationalize statements about validity by describing the relationship between (plausible or controversial) simulated empirical scenarios, (alternative) methods, and the impact on the results. We have different options to operationalize statements about validity:
>
> - **Supporting Validity**: We want to show that for all plausible simulations (i.e., those likely to correspond to the real scenarios) our method produces valid results.
> - **Threatening Validity**: We accept that for some controversial simulations (i.e., those unlikely to correspond to the real scenarios) our method produces invalid results.
> - **Invalidity**: We do not accept that for a plausible simulation (i.e., one likely to correspond to the real scenarios) our method produces invalid results.

Our paper encourages researchers to capture those kinds of statements in simulations operationally. This may happen upfront to research or during revisions.

Explicit, Operational Assumptions

Running the method of the empirical study in reverse, capturing all its assumptions in simulation code, is not as trivial as it seems. It leads to non-trivial insights, as we will show in the meta-validation section.

The simulation can 'hard-code' assumptions that are mostly implicit in the original method of a study. The simulation can be explicit on the process, given by the

direction of relationships, on how the data is sampled from a bigger population, on mechanisms that lead to missing data, on unobserved data like the results, or on the way how measurement works.

This creates an operational and potentially parametrized simulation of our assumed reality. For instance, a simulation may first simulate artificial data, and then simulate different mechanisms on how parts of the data get lost. This simulates problems with measurement. How a methods reacts to such data can then be examined operationally by running the simulation and afterwards the method on the produced artificial data.

Plausible and Controversial Simulated Scenarios

A simulated scenario can be considered as a complex but operational form of an assumption.

We iterate plausible or controversial simulations to strengthen the method by examining the impact on the results. Comparable to a regular assumption, everything derived in that way is conditional on the plausibility. In this case, it is the plausibility of the simulation that we deliver as an operational artifact. Rating if simulations are plausible or controversial is often subjective and can benefit from the involvement of reviewers. However, we now have an artifact that supports such review and discussion.

When designing, reviewing, or revising a method, the aim should be to stick to the most basic method, which is most resistant against the important threats operationalized by simulations. A catalog covering simulations of typical threats in MSR/ESE may help in such a case.

5.3.2 A Simple Example: Logistic Regression For Defects

We start with a simple example of how a simulation may operationalize statements on the validity of a method. We will check the application of a basic logistic regression model, comparable to the method in many past and recent studies on software defects (e.g., in [KSA+13, FBF+20, MW00, YXF+20, JTH21, NZZ+10, ZPZ07, ZN08, TMHI16]).

Technically, a simulation implements stochastic and functional relationships between variables. It draws variables from random number generators (stochastic) or produces variables according to basic mathematical functions (functional). Such simulations boil down to very basic code.

Understanding the following section does not involve specific libraries or extensive statistic background knowledge.

Original Method

A study that uses logistic regression to examine defects formulates a model that describes the relationship between some observed variable X, typically a software metric, and the observed defect Y. Both variables can be mined from repositories. The results of such method improve the understanding of the relationship, for instance, discussed in [KSA+13, FBF+20, MW00, YXF+20, JTH21, NZZ+10, ZPZ07, ZN08, TMHI16].

The most basic form of a logistic regression characterizes the relationship in terms of two unobserved variables, identified using the observed X and Y variable:

- **Intercept (alpha)**: The intercept is an unobserved variable reflecting the average probability of defects when $X = 0$.
- **Slope (beta)**: The slope is an unobserved variable reflecting the change in the probability of defects when X increases by one unit.

To exemplify this analysis, we borrow data from the *elasticsearch* project, published in the context of examining defects in [FBF+20]. We use a binary defect classification for our observed variable Y (computed according to SZZ [SZZ05]). The variable X is the stated-log transformed lines-of-code changed by a commit. For further details, we refer to the original study.

The code we use to invoke a logistic regression, in essence, an algorithm that identifies alpha and beta using X and Y, is given in Listing 5.1:

Listing 5.1: The original method applies a logistic regression to model the relationship between X and Y (R code).

```
1 model <- glm(Y ~ X, family = binomial())
```

When running this code, it reports that the unobserved intercept is -3.28 and the slope is 0.45. From the perspective of the results, the interesting aspect is the existence, the strength, the direction, and a potential causal nature of the relationship between changed lines-of-code and defects. Because of the positive slope (0.45), we may now conclude that commits with more changed lines are more dangerous, as this increases the probability of defects.

In such a trivial case, the method is often not further questioned. We trust in the fact that we apply the logistic regression correctly, that the software works, that we have enough data, and that our interpretation is valid, although we have never seen the real relationship, and the real values of the unobserved variables and compared them with our identified values -3.28 and 0.45.

We may implement additional checks by cross-validation. However, cross-validation is an important remedy against overfitting, it does not resolve the conceptual issue of not knowing the real intercept and slope.

Replacing the Empirical Scenario by a Simulation

We will now replace the real scenario with a simulation. In particular, this means that we replace some (or all) observed and unobserved variables with artificial counterparts, carefully simulated according to assumptions about the real empirical scenario. We will capture this in an operational simulation artifact.

We provide the simulation code in Listing 5.2 and as an online resource. Comments help to distinguish between scalars, vectors, and matrices. All simulation code in the thesis is written in standard R, not using advanced libraries.

Listing 5.2: The simulation artifact: R code that replaces data used by the original method (X and Y) with artificial data.

```
1  # Kept observed variables.
2  N <- N # Number of commits.
3  X <- X # (vector) Keep the original variable X.
4
5  # Substituted unobserved variables.
6  alpha <- -3.0
7  beta <- 0.4
8  prob <- 1 / (1 + exp(-(alpha + beta * X))) # (vector) Assumption of the logistic regression
       model on the relation between X and Y.
9
10 # Substituted observed variable Y.
11 Y <- rbinom(N, size = 1, prob = prob) # (vector) Assumption on the output distribution.
```

The first three lines of Listing 5.2 denote that we keep the original variable X and the number of observations N to stick close to the original data. However, this is not necessary. We could also rely on fully artificial data.

Next, the code assigns the unobserved intercept and slope variable in terms of *alpha* and *beta*. This is a crucial part of the simulation code, where the unobserved variables that correspond to our results, get replaced by artificial variables. We *invent* both values. We use slightly different values, compared to those that are the result of the original study run, to point out the fictional character of this replacement. We now use -3.0 for the intercept and 0.4 for the slope. We will test alternatives in the next section more systematically.

All the remaining code (line 8 and 11) directly follows the basic assumptions of a logistic regression on the relationship between X and Y (just run in reverse). MSR/ESE studies (e.g. [KSA+13, FBF+20, MW00, YXF+20, JTH21, NZZ+10, ZPZ07, ZN08, TMHI16]) make this assumption implicitly when using a logistic regression model; authors are aware of this definition.

The code specifies the exact probability *prob* of facing a defect as a (logistic)

function of *alpha*, *beta* and X. This vector of exact probabilities is another intermediate unknown that can never be observed. In reality, we can just observe the final defect classification Y. We simulate defects Y by a stochastic function producing uncertainty, a binomial distribution (*rbinom*) with one trial and the probability set to the vector *prob*. It is a simple random number generator[2].

Running the Original Method on the Simulated Data

We can now process the artificial data (X and Y) as if it were the real data, using the original method from Listing 5.1 with one important difference: *We know alpha, beta, and prob.*

Correspondence of the Results The original method identifies the unobserved variables (the results we like to interpret) in the simulated run to be $alpha \approx -2.97$ and $beta \approx 0.39$. We can objectively support the validity of the method under the simulation because the results *alpha* and *beta* are very close to those values set in the simulation (-3.0 and 0.4). However, there are two remaining questions on such correspondence.

Uncertainty of the Results First, it is unclear why the method does not exactly meet the artificial *alpha* and *beta*. It is because of the stochastic relationship used between *prob* and Y (Listing 5.2, line 11). The method only observes Y, but the corresponding *prob* is not known for sure; hence, also the identified values for *alpha* and *beta* are uncertain.

If repeating the simulation, using different initial seeds for the used random number generators, the identified *alpha* and *beta* distribute as depicted in Figure 5.1. This shows that on average, the identified alpha and beta variables are excellent, but not totally exact. This is a reason many studies report on confidence intervals and p-values.

Parametrized Tests Second, it is not clear what happens if we use different artificial values for *alpha* and *beta*. The simulation can examine this by going through multiple combinations of both, checking the impact of the method on its results.

The results for a grid of 30×30 combinations is shown in Figure 5.2 (left). The plot illustrates the impact of a particular combination of our assumptions, operationally given by the artificial alpha and beta, on the difference between the identified and the artificial beta. Such a difference can be interpreted as an error in the identification done by the method. Figure 5.2 (left) shows this as a scatter plot. The assumptions in terms of artificial alpha and beta can be read from the two axis, and the error is depicted by the red color of a dot. When alpha and beta are both high or low, identification struggles, it produces a high error (red dots). On the diagonal, e.g., when alpha is high and beta is low, errors are low (white dots). We do not show the error in the identification of alpha, but a plot can be derived in analogy.

[2]In R, random number generators are vertorized and start with a letter *r* followed by an abbreviation for the distribution family (we will see *rbinom*, *rnorm* and *rpoisson*).

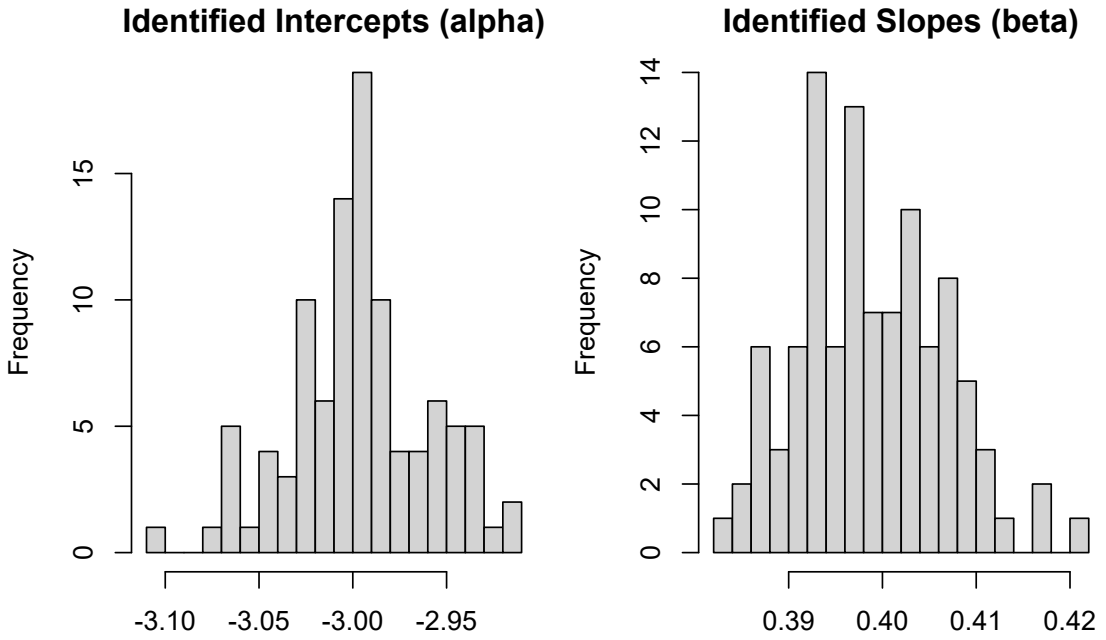


Figure 5.1: Identified unobserved variables intercept (alpha) and slope (beta) by the method in 100 repeated simulation runs.

The phenomenon is well-known. If counting the number of defects in the artificial Y variable and relating it to this error, we see that the identification only struggles on extremely high or low defect counts (see right of Figure 5.2).

We know this under the name *class-imbalance* [TTDM15] or in terms of instructions on events-per-variable (EPV) [PCK+96]. Our simulation does detailed suggestions, i.e., that if we have more than approximately 400 defects in the data set, the method should be safe from the threat of too low defect counts. In our real data set, we have 5771 defects, so we can deny the plausibility of this simulated scenario, and thereby also the impact on the error in results.

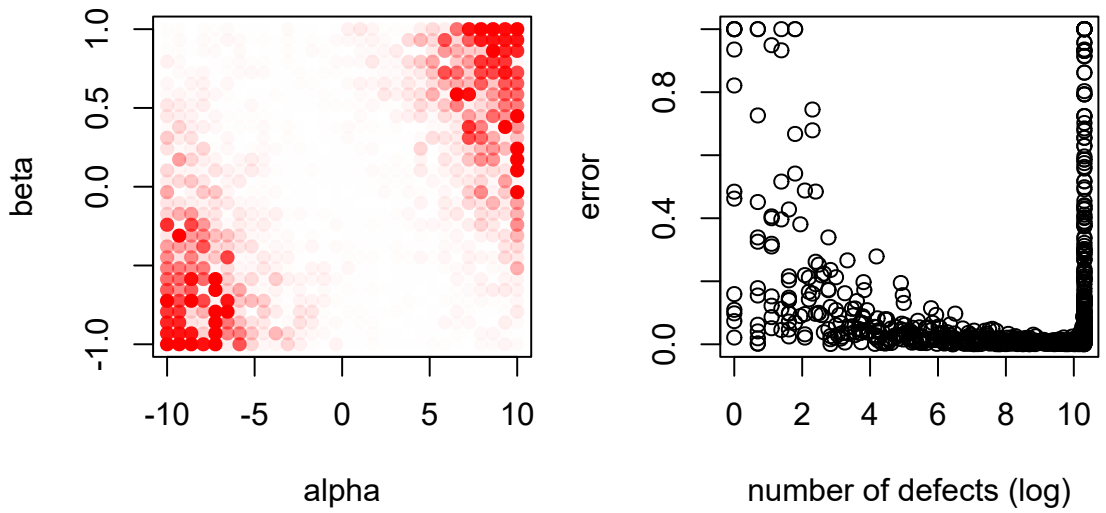


Figure 5.2: Dots are repeated simulation runs. **Left:** Artificial alpha and beta and the error in the identification, depicted as red dots (red increases with error). **Right:** The total sum of defects in the artificial variable Y (log scale) and the relation to the error in the identification.

5.4 Evaluation

In this section, we present an evaluation by applying simulation-based testing to existing studies in MSR/ESE.

5.4.1 Meta-Validation

To provide a validation on our part, we apply simulation-based testing to six real scenarios that are studied in MSR and ESE. We show what benefit can be expected by simulations as additional validation artifacts. We show that we can either i) support validity, ii) threaten validity, or iii) show the invalidity of the used methods and results by simulated scenarios.

Section Structure

The following six sections will provide this meta-validation of simulation-based testing (Sec. 5.4.2-5.4.7). These sections follow the structure:

- **Research Question**: We instantiate our meta research questions for a particular empirical study. We discuss how we will answer the research questions in the concrete case by a simulation artifact.
- **Original Method**: We describe the original method of a study (or of a type of study) that may or may not have problems with validity.
- **Simulated Scenarios**: We describe how we replace the assumptions about an empirical scenario, subject to the study, by one or more simulations of the scenario. The simulation artifacts are provided online.
- **Rating Results**: We discuss the results provided by the original method when applied to the simulated scenarios.
- **Conclusions on Validity**: We conclude on the validity of the original method conditional on the plausibility of the simulations.
- **Revision (Optional)** We show how to improve the original method so that it produces correct results for the simulated scenarios.

Summary

We give a short sketch of the **main findings and argumentation patterns** for the cases examined in this meta-validation.

- **Dependent Observation (Sec. 5.4.2, Invalidity)**: We show that a used method computes results with incorrect confidence intervals in a simulated scenario where observations are dependent. Especially in MSR, observations are often sampled from the same repositories which may introduce dependency. This is a plausible scenario, and thereby may invalidate parts of the original study.

- **Prediction or Causation (Sec. 5.4.3, Supports Validity)**: We show that when using a logistic regression method, the results can also be interpreted causally if the underlying simulated scenario is sufficiently basic. The simulation supports the validity of the original study. We hint at limits and potential improvements of the plausibility of the proposed simulation. We explain how extending simulations may also lead to simulations that threaten claims on causation.
- **Control of Variables (Sec. 5.4.4, Invalidity)**: We show that under a plausible simulation, a method to prove the relevance of a novel metric, produces trivial and misleading results. This renders the method as invalid. We show how to revise the method.
- **Correlated Variables (Sec. 5.4.5, Invalidity)**: We show that under a series of plausible simulations, a new method always fails to produce correct results. No simulated scenario shows an improvement of the results. In some scenarios, the quality of results even decreases. This renders the proposed method as invalid.
- **Distribution Types (Sec. 5.4.6, Supports Validity)**: We show that a certain type of mismatch between method and simulated scenario does not influence a certain interpretation of the results. The interpretation of the original study is still valid. We support the validity.
- **Experimental Research (Sec. 5.4.7, Supports Validity)**: We simulate a random experiment and show the impact of some alternative methods on the interpretation of results. We support the validity of such studies.

Acknowledgement

We want to acknowledge the original work of the authors in the studies, subject to the following illustrations. All studies have been selected because of their originality. However, we believe that this meta-validation of simulation-based testing is not credible on unpublished examples. We need to continue on real studies.

5.4.2 Dependent Observation (Case 1)

After laying the foundations, we start with the first case for our meta-validation. We examine a study by Alali et al. [AKM08]. The work examines the 'typical commit'.

Research Question

The original paper examines a research question, simplified as follows:

- **RQ***: What is the typical commit in a software repository?

In this paper, we are interested in operationalized statements about the validity of the way the study attempts to answer this research question. We instantiate our meta research questions for the study accordingly:

- **RQ*1**: What assumptions of this study on repositories, commits, and properties of the typical commit can be operationalized?
- **RQ*2**: What is the impact of such assumptions on the result of the study regarding the properties of the typical commit?

We believe that most results of Alali et al. are valid. However, the study runs into a specific problem, that we believe, is characteristic for the analysis of repository data. The results include statements on uncertainty, while the method ignores a potential structure in the analyzed sample of commits.

We will provide simulation artifacts that operationalize these assumptions, and then show that they invalidate confidence intervals for the results.

Original Method

The original method of the study mines the history of nine open-source software systems. It reports on the number of lines, files, and hunks changed by commits. Further, the correlation between the variables is described.

Examining the 'typical commit' indicates that the resulting statements are not necessarily specific to the nine observed repositories. Statements may also hold for other repositories, those that the authors had in mind, but did not examine due to computational overhead. Such sampling is standard practice in MSR [CIC16].

A method needs to produce results with a notion of uncertainty because variables identified using the nine repositories might not exactly correspond to values computed including the unobserved repositories. A method can give such statements in terms of confidence estimates.

The method of Alali et al. report on confidence, i.e., p-values for the correlation between the variables. In the remainder, we will leverage instead the notion of confidence intervals, since p-values are less intuitive for most readers. Problems with p-values are the same and can be shown the same way.

We assume that the original method computes p-values according to standard practice because the paper does not report on counteractions against dependent observations. In our reproduction of the original method, we also stick to standard practice[3].

Simulated Scenario A

The data of [AKM08] is not available, so we simulate all the data of the empirical scenario. We simplify the original research and examine the correlation between just two variables $X1$ and $X2$.

Listing 5.3 shows simulated scenario A, which simulates two correlating variables in a structured sampling process.

Listing 5.3: Simulating two correlating variables X1 and X2 for 100 repositories, each with 100 commits.

```
1 X1all <- NULL # X1 collected over repositories.
```

[3]All our reproductions of other papers are fully available online to guarantee the reproduction of this thesis.

```
X2all <- NULL # X2 collected over repositories.
N <- 100 # Number of repositories.

for (repo in 1:N) {
    rho <- 0.2 # Rho is the same in each repository.
    M <- 100 # Number of commits in each repository.
    # Simulating X1 and X2 for a repository.
    X1 <- rnorm(M, mean = 0, sd = 1)
    X2 <- rnorm(M, mean = 0, sd = 1)
    # Producing the correlation (rho).
    sigma <- matrix(c(1, rho, rho, 1), 2, 2)
    X <- cbind(X1, X2) %*% chol(sigma)
    # Collecting X1 and X2.
    X1all <- c(X1all, X[, 1])
    X2all <- c(X2all, X[, 2])
}
```

The code produces data for N=100 repositories, each with M=100 commits. Within a repository (inside the loop), we simulated two variables $X1$ and $X2$ for M commits following a normal distribution ($rnorm$). A correlation between $X1$ and $X2$ is simulated using the *Cholesky decomposition* with $rho = 0.2$.

Finally, we simulate the random sampling step (see the online resources for the complete code). We randomly decide on 91 repositories where we consider $X1$ and $X2$ as unobserved variables, and 9 repositories where we consider $X1$ and $X2$ as observed variables.

Simulated Scenario B

The assumption of dependent observations is added by a small modification to simulated scenario A.

Listing 5.4: Simulating a repository-specific variation in the correlation rho.

```
for (repo in 1:100) {
  rho <- rnorm(n = 1, mean = 0.2, sd = 0.23) # A repository-specific variation in the
      correlation.
  # ...
}
```

In simulated scenario B, the correlation rho is sightly different for each repository, but on average 0.2, since we draw it from a normal distribution with mean 0.2. The standard deviation (sd) decides on the severity of the threat, we set it to 0.23, but other configurations can be explored in the same fashion as done in Sec. 5.3.2. Some values that cannot serve as a correlation rho need to be filtered out. The rest of the simulated scenario is the same as in the previous simulated scenario A.

Rating Results

The original method of Alali et al. [AKM08], used to report on the uncertainty of the results on the typical correlation, works under simulated scenario A, but not under simulated scenario B. Repeating ten simulation runs suffice to show this.

We compare the correlation (rho) computed on all 100 repositories, including the unobserved, with the confidence interval identified according to the original method

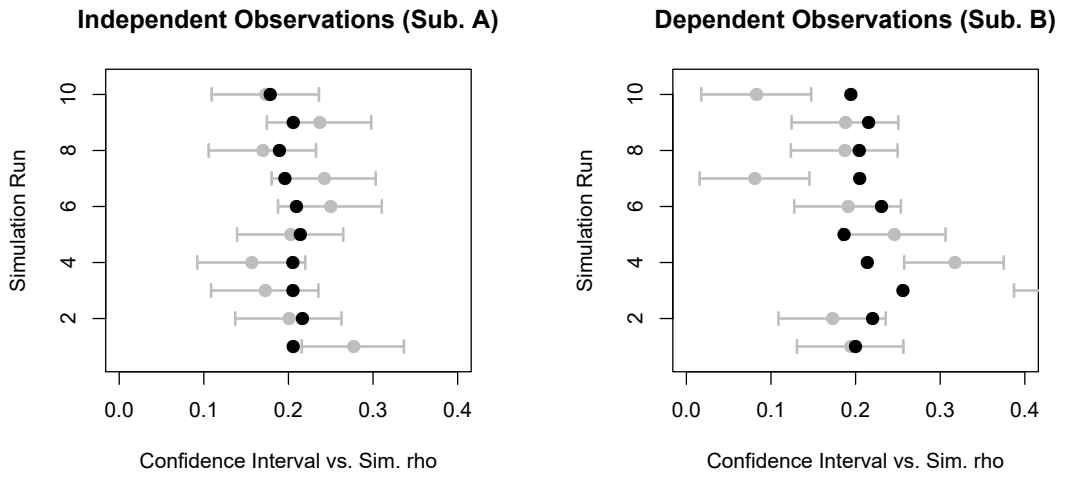


Figure 5.3: Ten simulation runs showing confidence intervals for rho, identified based on 9 repositories (gray line), and rho computed on all 100 repositories (black dot). We distinguish between independent (left) and dependent observations (right) produced according to simulated scenarios A and B.

on the 9 observed repositories. Since we report on confidence intervals of 95%, 0.5 out of 10 simulation runs are allowed to fail. When having a look at Figure 5.3, we see that the number of failing confidence intervals is different in both simulated scenarios:

- Under *simulated scenario A* (left), almost all confidence intervals include the correct correlation.
- Under *simulated scenario B* (right), we see that in 4 out of 10 simulation runs, the method fails to include the correct correlation in the confidence interval. The small variation regarding repositories has a big impact for statements on uncertainty. The chance that our confidence interval includes the correct correlation, is almost comparable to flipping a coin.

Conclusions on Validity

We have operationally defined the assumption of dependency in observations, and then we have shown that it may invalidate the results produced by the original method regarding uncertainty.

Indicators for the plausibility of simulated scenario B, compared to simulated scenario A, can be found in every analysis that proves that repositories are different (e.g., see the highly different slopes in regression models computed on different repositories in [FBF+20, KSA+13]). We may even check such assumption on dependency on the data (which we don't have in this case).

The impact of our assumptions about the number of repositories N, and the standard deviation sd can be explored by a parameterization of the simulation. This may uncover that more repositories and lower standard deviations decrease the error in computed confidence intervals.

Revision

General advice on improving the method can be guided by informing the model of the structured sampling process. In the simulation above, we simplified. We recommend casting the problem as a linear model, where slopes correspond to the correlation we like to examine. The linear model can be evolved to a *hierarchical linear model*, where the repository-specific impact on the correlation can be identified as a random effect. The random effect may follow a normal distribution and the standard deviation is thereby identified, too.

We recommend testing such models in simulations. We recommend work on hierarchical/multilevel/mixed-effect models for guidance [GH06].

5.4.3 Prediction or Causation (Case 2)

The next case in our meta-validation examines an experience report by Tantithamthavorn et al. in [TH18]. The report discusses challenges and actionable guidelines when using methods for defect modelling.

We selected this work because of its progressive understanding of defect modeling to be more than just defect prediction. We expect this report to have a big impact on our community. However, this work fears to name one actual challenge, which is *examining causation*. We will show how to sharpen this understanding by simulations.

Research Question

The original paper examines a research question, simplified as follows:

- **RQ***: What are the challenges and actionable guidelines when using methods from defect modeling?

We are interested in operationalized statements that complement such guidelines. We instantiate our meta research questions for the study accordingly:

- **RQ*1**: What assumptions of this study on causes for defects in commits can be operationalized?
- **RQ*2**: What is the impact of such assumptions on the result of the study when talking about causation?

We will accomplish the original study by simulation artifacts that support the validity of claims on causation. This sharpens the guidelines of Tantithamthavorn et al.

Original Method

Practitioners may use an analytical defect model not just to predict defects, but also to answer questions, like *'whether complex code increases project risk'* (copy from [TH18]). Project risk refers to defects.

Within this setting, the term *increases* can be understood in different ways, and the original study is unclear on this:

- Do we wish to compare or relate complex code with project risk? If so, we can use this insight to (mentally) predict one variable using the other.
- Do we wish to know if modifying the complexity of code causes the project risk to change? This insight is efficient to guide our decisions. It corresponds to what most people have in mind when hearing the statement above.

Tantithamthavorn et al. and many other researches in MSR/ESE fear to claim causation (second statement). This is not surprising. Causation is difficult to examine, definite claims are impossible, even in randomized experiments.

On the other hand, most of the operational decisions should be driven by causal relationships. Their examination is the crucial point of modern empirical research (see the preface of [IR15]). Even more dramatically, statements on the relation between *complex code* and *project risk* are intrinsically hard to understand if not aiming at causation. Understanding may get harder if a defect model grows, without having causation in mind, including more variables.

Simulated scenarios are a useful extension to the original paper, since they can make clear when claims on causation are valid.

Simulated Scenario C

Following the definition of Imbens et al. [IR15] (page 6), a claim on causation can unambiguously be given by comparing the potential outcomes of different treatments in exactly the same situation. Reality does not allow observing more than one potential outcome in the same situation, but a simulation-based test allows synthesizing both.

Listing 5.5: Simulating causation.

```
X <- rnorm(N) # Synthetic variable X.
# Producing two potential probabilities.
prob_pot1 <- 1 / (1 + exp(-(alpha + beta * X)))
prob_pot2 <- 1 / (1 + exp(-(alpha + beta * (X + 1))))
# Corresponding potential defects.
Y_pot1 <- rbinom(N, size = 1, prob = prob_pot1)
Y_pot2 <- rbinom(N, size = 1, prob = prob_pot2)
```

Listing 5.5 implements such simulated scenario, using 'treatment' X, but also continuing with the modified $X+1$, in exactly the same situation. The relationships are the same as in the basics on logistic regression provided in Sec. 5.3.2. In the simulation, we get two potential outcomes. According to the definition, the difference between Y_{pot1} and Y_{pot2} reflects the causal relationship between X and Y.

Rating Results

According to our previous strategy of hiding artificial but unobserved variables from the original method, we run the original method just using one of the potential outcomes, keeping the other hidden. However, the simulated scenario shows that we can identify the causal relationship by the logistic regression method.

Multiple runs of the simulation, with different artificial values for *beta*, show that there is a clear correspondence between the identified *beta* using only the ob-

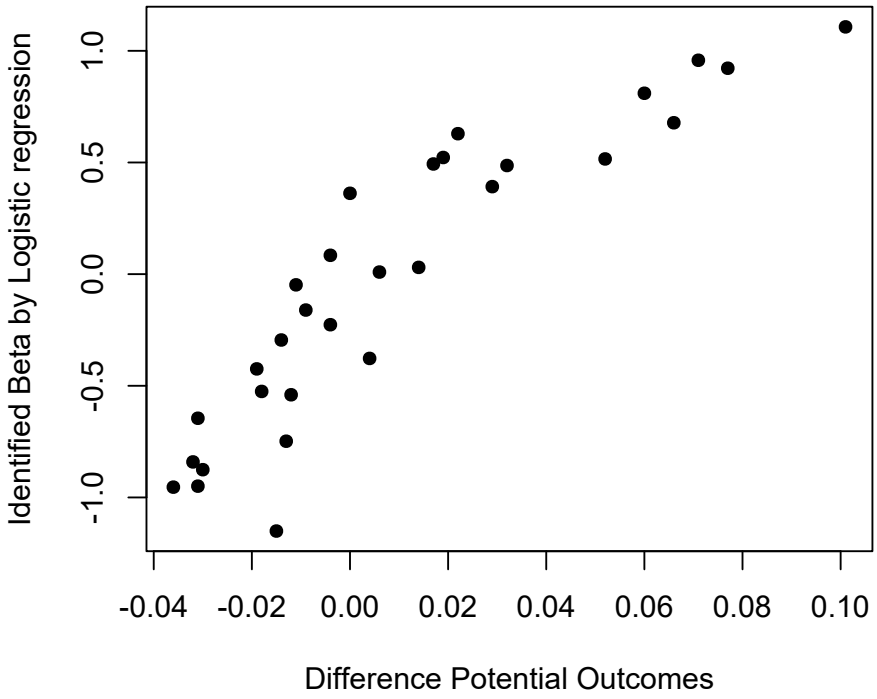


Figure 5.4: Identifying causation under strong assumptions (simulated scenario C).

served variables, and the difference between both potential outcomes also including unobserved variables (see Figure 5.4).

Conclusions on Validity

We have provided operational assumptions about a scenario on causes for defects, and did show the impact of such a scenario for claims on causation in the context of the method examined in the original study.

Our simulation artifacts provide a clean notion of assumptions and the implications on the validity of claims on causation. Our simulation artifacts can be used to sharpen the guide by Tantithamthavorn et al., being more precise on what the *interpretation of defect models* is.

However, simulated scenario C may not be considered as the most plausible scenario. In fact, it is very limited. It assumes the effect of X (*beta*) to be stable across all our observations; no dependency between observations; X is just a random variable, not influenced by anything else. The simulation may be evolved based on these insights, and the results of methods can be checked if they are valid claims on causation. This may render some threats to claims on causation that we prefer to make explicit.

5.4.4 Control of Variables (Case 3)

In [TBP+17], the authors propose a new metric to reflect a developer's ability to correctly understand the code.

Research Question

The original paper examines a research question, simplified as follows:

- **RQ***: What is a metric that reflects a developer's ability to correctly understand code?

We are interested in operationalized statements about the validity of the method used to prove the relevance of the new metric. We instantiate our meta research questions for the study accordingly:

- **RQ*1**: What assumptions of this study on a new metric regarding the developer's ability to correctly understand code can be operationalized?
- **RQ*2**: What is the impact of such assumptions on the result of the study, which is aimed at correctly proving relevance?

We show simulations for which the used method always proves relevance. The simulation uncovers that the used method is incapable of producing negative results. This renders the method as invalid.

Original Method

The original method follows the standard in defect modeling as described in Sec. 5.3.2. It relates the novel experience metric E to defects Y. Defects are used as a proxy for a developer's ability to correctly understand the code. The method is slightly adapted because the authors show the relation using a basic statistic test, rather than a logistic regression. The proof of the existence or non-existence of such a relationship can be considered as the result of the method.

The novel experience metric is defined using the cosine similarity between *files* touched by a commit, and the lexical background of the contributing developer (*back*). Defects may increase, as similarity decreases. This lexical background is composed out of all modifications done by the developer in the past. We refer to the original work for details on how to compute the exact metric.

Our first intuition reading the original paper was that the technical computation using the cosine may accidentally influence the results of the method. We expected a potential correlation between the cosine and the file size.

This is a serious threat if we aim to interpret the fact that it is experience, and not the file size, that influences defects. We know from works in MSR/ESE, e.g., [Moc10, FBF+20, KSA+13], and from Sec. 5.3.2, that the file size (or lines-of-code) strongly relates to defects. Such *confounding effect* of the new experience metric over variable *files size* would not be surprising and not much of interest. The new metric would just be an overcomplicated proxy for file size.

Simulated Scenario D

This means that the method used in the study needs to protect against such false claims on the existence of a relationship. We can write a simulation-based test to check such a protection. This scenario simulates a part of the metric computation to produce variable X and E using the cosine.

Listing 5.6: Simulating the computation of experience.

```
N <- 8000
X <- NULL
E <- NULL
for(n in 1:N){
  nTerms <- 200
  # Generate two random term vectors.
  back <- rpois(n = nTerms, lambda = 5.0)
  file <- rpois(n = nTerms, lambda = 0.1)
  # Compute the similarity defining experience.
  E <- c(E, cosine(back, file))
  # Size of the file.
  X <- c(X, log(sum(file) + 1))
}

alpha <- -3.0
beta <- 0.4
prob <- 1 / (1 + exp(-(alpha + beta * X)))

# Substituted observed variable Y.
Y <- rbinom(N, size = 1, prob = prob)
```

Listing 5.6 produces N artificial file and background pairs using vectors sampled from a Poisson distribution (stochastic function). The number of terms ($nTerms$) in the VSM is set to 200. The Poisson distribution works well for simulating term vectors because it only produces discrete positive vector entries. The average term frequency is set by the *lambda* parameters. The code collects the new experience metric E defined by the cosine, but also the corresponding file size X as the stated-log transformed sum of its terms (as often assumed in defect modeling). Alternatives can easily be explored using the online material. According to our knowledge, they do not influence our conclusions under simulated scenario D.

Finally, the code simulates the probability *prob* and the defects Y, as we have done in the previous sections. In this simulated scenario, we assume that there is no effect of E. This is clear from the code because E is not input to the function producing *prob*.

Rating Results

The original method does not manage to handle the simulated scenario D correctly. The Mann-Whitney test used in the original study rejects the null-hypothesis (which it should not do according to our assumptions of the simulation) in approximately 45 out of 100 simulation runs at a confidence level of 95%. That means that in 45 runs, it incorrectly detects the existence of a relationship. Doing it in 5 runs would correspond to the confidence level. Hence, the result is invalid.

The reason gets clear when looking at the raw data in Figure 5.5, showing a strong positive correlation between X (file size) and E (novel experience metric). The very simplistic Mann-Whitney test accidentally attributes the effect of X (file size) to E (novel experience metric). As expected, it is caused by the technical computation of the cosine.

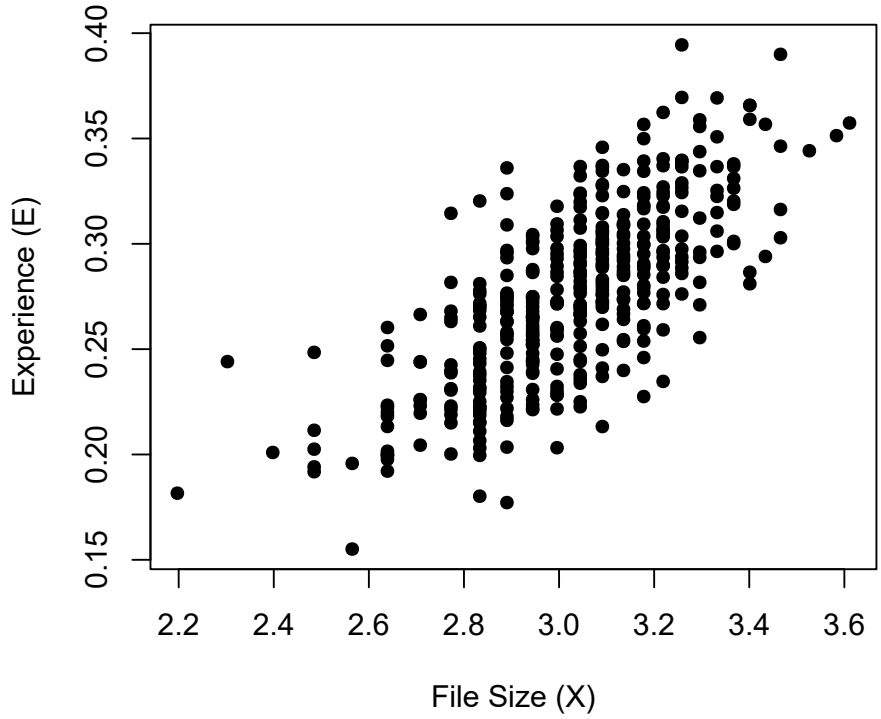


Figure 5.5: Relation between file size X and experience E computed by the cosine in a single simulation run.

Conclusions on Validity

The previous simulated scenario operationally shows how a method is technically not suited to produce valid results. We show this in an important case for the researchers, proving their new contribution to be wrong.

Simulated Scenario D is plausible enough in that we can demand a method to resolve it. Claiming that a developer-specific factor relates to defects, while it is essentially just the file size, does not provide a novel insight. This holds for examining causation as well as for prediction.

Revision

Indeed, there are ways to improve the original method. We appreciate that the original data is provided by the authors so that we can rearrange the statistical checks.

We convert the original test (Mann-Whitney) into a logistic regression model, which allows the *control of variables*. The control for the variable *file size* is the mandatory step that resolved the threat to a method. It blocks the *confounding effect* of the new experience metric over file size, in that we get the *direct effect* of experience that we are interested in. Such a model indeed succeeds in *not detecting* an effect on the artificial data produced by simulated scenario D. See the online resources extending the simulation for this insight.

We now apply the method (with and without control) to the real data and revise the original study. The relevant information on the logistic regression models can be found in Table 5.1. For further details, we refer to the reproduction code.

Table 5.1: A model with and without control applied to the real data, showing the effect strength, the usual significance encoding, and AIC.

Variables	Original	Control
Experience	1.42^{***}	0.12
File Size		0.01^{***}
AIC	19968	18282

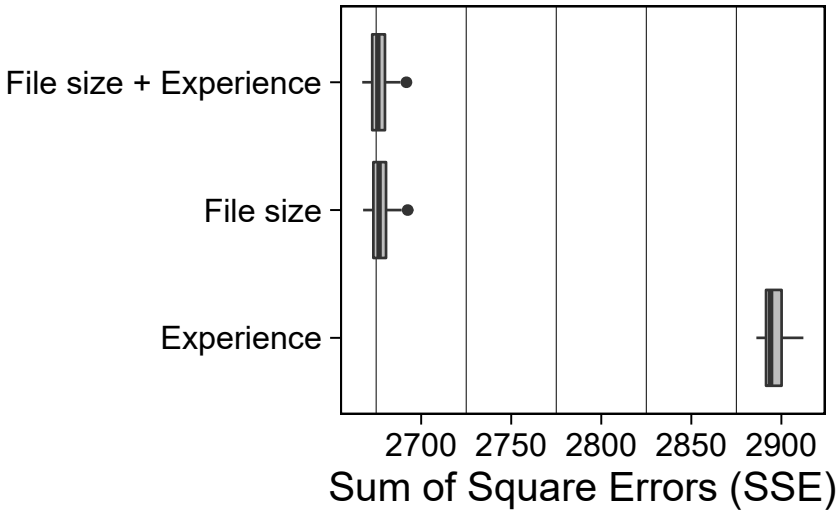


Figure 5.6: The performance impact of the new experience metric evaluated in cross-validation on the original data.

- We start with a reformulation of the original statistic test, describing the effect of *experience* on defects as a basic logistic regression. This model is called **Original** and reports that, comparable to the original work, the effect of the novel experience on defects is positive and highly significant.
- The second model, called **Control**, adds the file size as a control metric. The effect of the experience drops by a factor of ten (from 1.42 to 0.12). The effect stops being significant. This conforms to our initial intuition that the study just proves a *confounding effect* over file size and not the importance of a new metric.

Even when evaluating the new metric in prediction, adding it next to file size, as a new predictor, does not lead to an improvement. Cross-validation results for prediction on the original data can be found in Figure 5.6.

A simulation would have discovered this problem with the computation of the cosine and the invalid method early.

5.4.5 Correlated Variables (Case 4)

The next study that we examine is by Jiarpakdee et al. [JTH21]. The study aims at showing a general improvement to methods interpreting defect models with correlated variables. The authors motivate the practice of removing variables based on correlation or VIF thresholds (VIF [CCWA13]).

Research Question

The original paper examines a research question, simplified as follows:

- **RQ***: Can we improve the results of a method for defect modeling by a threshold-based removal of correlated variables?

We are interested in operationalized statements about the validity of this improvement of the method. We instantiate our meta research questions for the study accordingly:

- **RQ*1**: What assumptions of this study on the typical interpretations of results and strong correlation between variables can be operationalized?
- **RQ*2**: What is the impact of such assumptions on the result of the study, which applies the new proposed variation of the method?

We will show that the proposed improvement to the method does not improve the results in a series of plausible simulated scenarios. In some scenarios, it even causes additional problems. We are not aware of a simulation that supports the validity in terms of showing that the improvement to the method provides better results. The new method can be considered as invalid.

Original Method

In a nutshell, the original study claims that model interpretation methods, that are run on data sets where correlated variables have been removed by thresholds, provide better results. Strongly simplified, Jiarpakdee et al. support this claim by showing that both methods provide different results. We refer to the original paper for the details on how the authors assume this argumentation to work.

Simulated Scenario E

We will simulate a series of scenarios with strong correlation, and check if a method that removes correlated variables leads to better results.

In the following, we focus on a fully artificial data set with the variables X, Z, W and the resulting defects Y. We will simulate a causal pattern where W is a confounding variable for the relation between X and Y, while Z and W may get strongly correlated depending on a simulation parameter.

Listing 5.7: Simulating relationships, producing correlated variables and defects.

```
# Alternative standard deviation of Z produces different correlation strength between Z and W.
for (sdZ in seq(0, 1, length.out = 40)) {
    # Stochastic relationships between W, X and Z.
    W <- rnorm(N)
    X <- rnorm(N, mean = -W, sd = 1)
    Z <- rnorm(N, mean = W, sd = sdZ)

    prob <- 1 / (1 + exp(-(W + X)))
    Y <- rbinom(N, size = 1, prob = prob)
    # ...
```

In this simulation, no variable influences W. Variable W influences X and Z. Both variables X and Z are given as stochastic functions following a normal distribution, with the mean set to be W or $-W$. Further, the stochastic function simulating Z is configured using different values for the standard deviation. This means that with decreasing standard deviation sdZ, the variable Z becomes a perfect copy of W.

The final defects Y are produced as a stochastic function of X and W. The variable Z is not related to defects.

Rating Results

We want to cover two 'interpretations': First, we are interested in the effect of Z and W. Both effects are unobserved variables that we set in the simulation. When running the logistic regression including all variables, the identified effect of Z and W is correct until the correlation reaches a threshold of about 0.9 (see Figure 5.7).

The improved method of Jiarpakdee et al. should resolve this. We expect it to drop Z, since the variable Z is not related to defects according to the simulation. However, this insight cannot be made by using the correlation or VIF values, since both are symmetric. A selection would be comparable to flipping a coin.

In a second 'interpretation', we are interested in the effect of X. We show the identified effect of X in models, including variables Z, W, none and both in Figure 5.8. The model not including W and Z fails as it runs into the problem with confounding. The model, including all variables, succeeds like the model including W. The model using Z fails up to the point that the correlation gets so high in that Z can be used as a replacement for W.

We see that neither dropping W nor Z makes sense, as the model including both does a perfect identification of the effect of X. If we decide between W or Z, we risk to drop the wrong variable.

Conclusions on Validity

The previous simulated scenario operationally shows how a proposed improvement to a method does not improve results.

The simulated scenarios that we show include a very plausible causal pattern. We can expect a study that recommends dropping correlated variables, with the aim of improving the interpretation of defect models, to exactly resolve such issues. We do not expect the results of a method to get worse.

We are not aware of any simulated scenario where the recommended practice brings real benefits for results. Even for prediction, dropping correlated variables just decreases performance (corresponding simulations are straightforward to implement). This conforms to recent statistic guidelines (e.g., see [McE20], page 169).

5.4.6 Distribution Types (Case 5)

The next example will focus on one of our previous works that is presented in [SHL+19]. We will illustrate the implications of choosing the wrong output distribution for a regression model. It is a mistake that we did in [SHL+19].

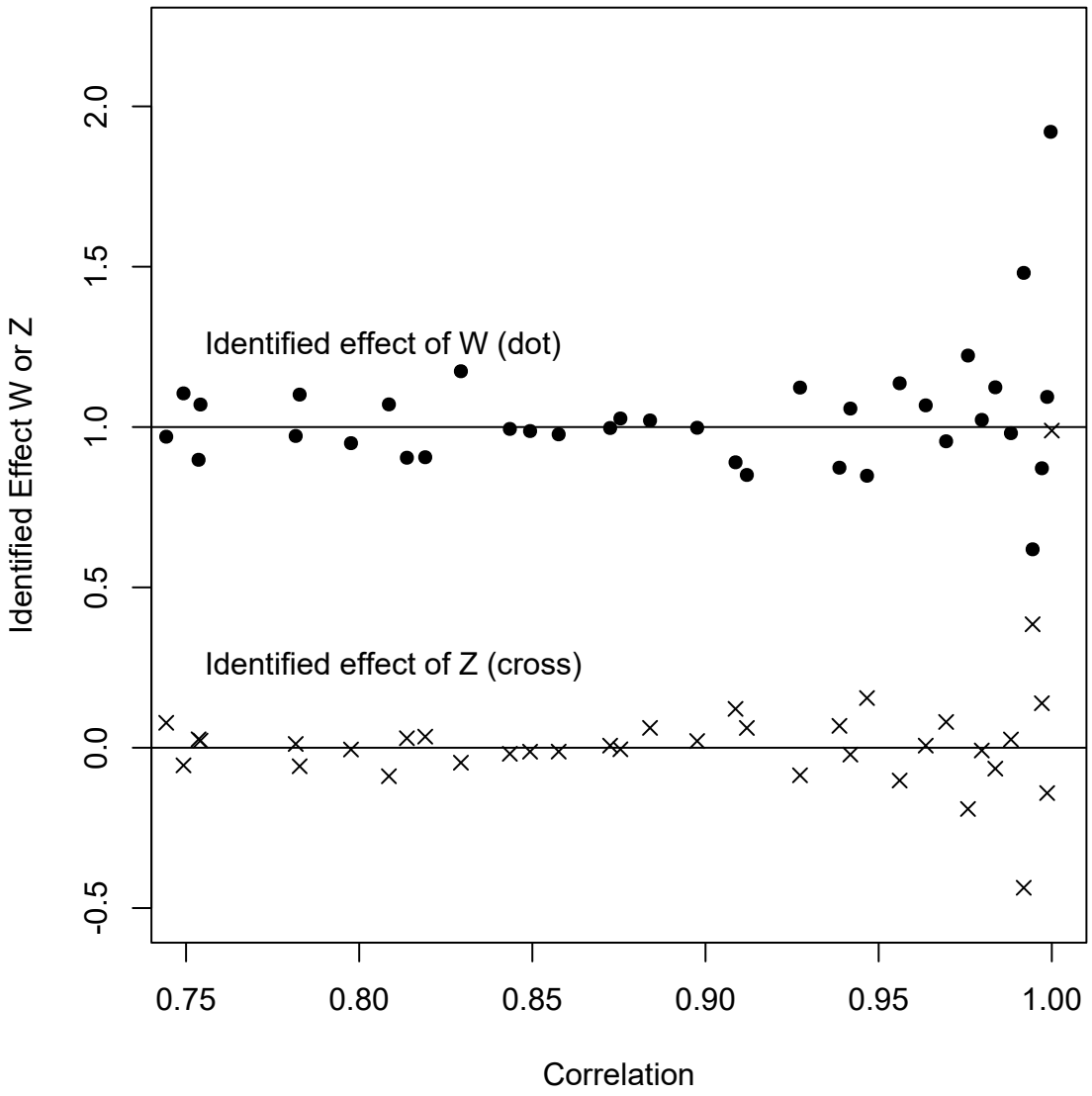


Figure 5.7: The identified effect of W and Z (which should be 1.0 and 0.0 respectively) under different correlation.

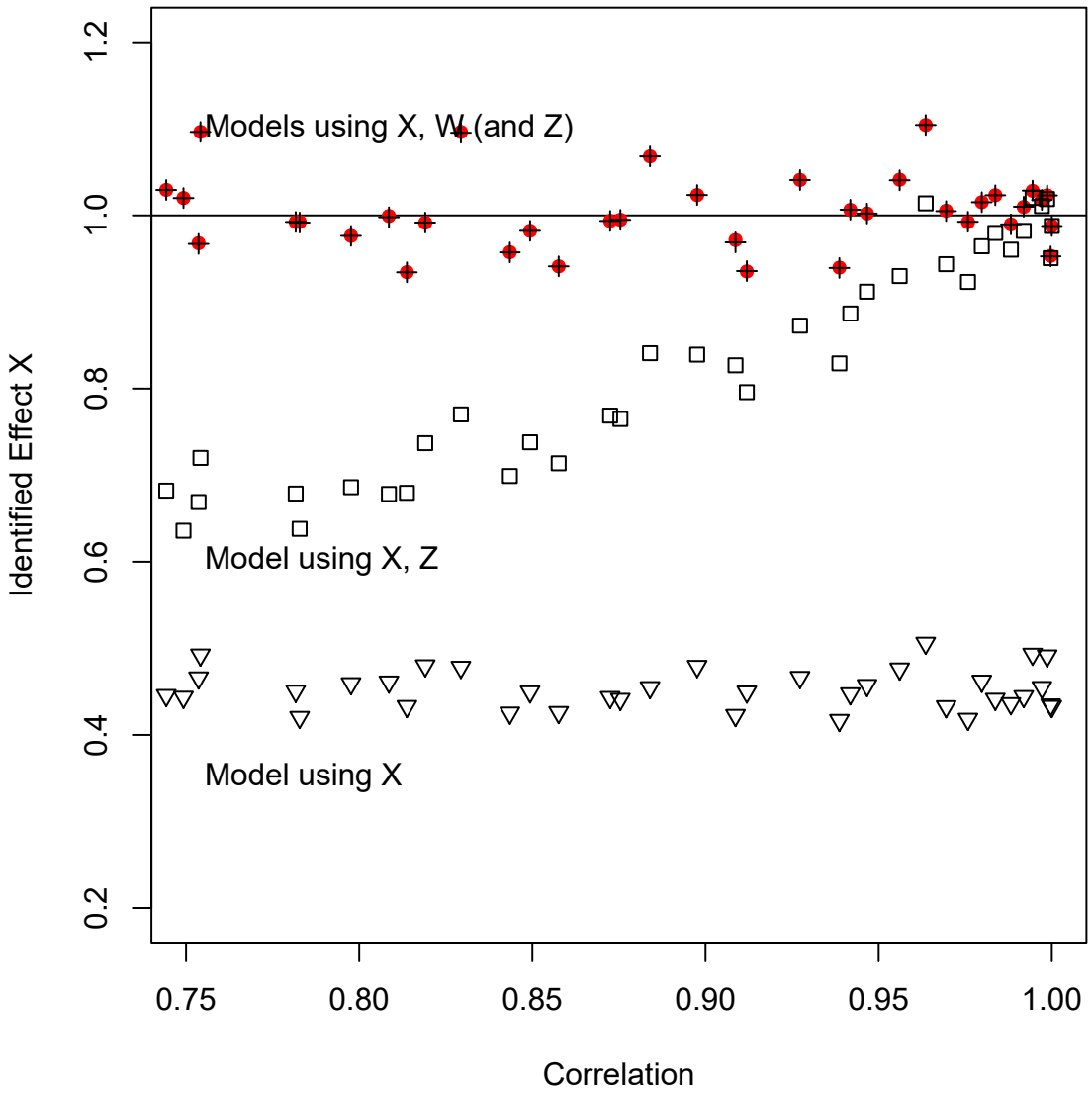


Figure 5.8: Models in simulation runs and the identified effect of X (which should be 1.0) under different correlation and different control variables.

Research Question

The original paper examines a research question, simplified as follows:

- **RQ***: What are the characteristic differences of repositories using SPARQL and Cypher queries?

We are interested in operationalized statements about the validity of resulting statements. We instantiate our meta research questions for the study accordingly:

- **RQ*1**: What assumptions of this study on the differences between repositories using SPARQL and Cypher can be operationalized?
- **RQ*2**: What is the impact of such assumptions on the result of the study, regarding statements about the difference?

In this case, we show how parts of the results of an obviously wrong method can still be valid under plausible assumptions.

Original Method

We will only focus on the parts of the method that run a regression model in this section. Other elements of the method should also be reviewed within the framework of the new strategy.

Originally, we used the regression model to better understand the decision of a software project between two alternative graph query languages, SPARQL, and Cypher. The model tries to associate the decision between the two languages with different properties of a project, i.e., with its age (*created_days_ago*), the popularity (*stargazer_count*), the number of active developers working on graph queries (*active_developers*), and the active files that include graph queries (*active_files*). In essence, the study results suggest that SPARQL is preferred by projects that are older, more popular, more active, and that have more files including graph queries.

In this section, we particularly focus on the output distribution of such a model, which needs to reflect the decision between SPARQL and Cypher. It thereby conforms to the binomial distribution with a single trial, where one language (e.g., SPARQL) is represented by 1 and the other by 0. The model thereby aligns with our previous discussion of defects.

Table 5.2: Parameters identified by different models that describe the decision for SPARQL: The table shows the difference in identified parameters when using a linear or a logistic regression model (normal vs. binomial output distribution).

Output Distribution Type	Normal	Binomial	Normal	Binomial
Variables	Parameters		Scaled Parameters	
created_days_ago	0.040	0.187	0.935	0.860
stargazers_count	0.017	0.086	0.392	0.393
active_developers	0.117	0.498	2.731	2.287
active_files	0.060	0.515	1.405	2.365

Due to a lack of experience with such modeling practice at the time, we decided to go for what, we believed, was the more established method. We used the wrong output distribution. We used a linear regression and thereby a normally distributed output. This is clearly a mistake. We provide the results of our original method on the original data, and a method using the improved binomial output distribution type, in Table 5.2.

The middle part of the table illustrates our mistake. We can see that the identified parameters differ for a model with normal and binomial output. For instance, the variable *created_days_ago* has an effect of 0.040 for a normally distributed output, while for a binomial output, we have an effect of 0.187.

However, our interpretation of the parameters, i.e., if a variable has a positive or negative effect on the decision for SPARQL, does not change. For instance, older projects with a higher variable *created_days_ago* still seem to prefer SPARQL. This consistent interpretation gets more obvious by scaling the identified parameters of both models, dividing them by their standard deviation. On the right side of Table 5.2, showing the scaled parameters, we notice that the results of the different models are very close to each other. Hence, if ignoring the scale, our interpretation of them still appears to be almost correct.

Simulated Scenario F

We will examine this in detail, to show that it is the regular behavior, if making this mistake, and not a fortunate coincidence. We start with producing a fully artificial version of the problem.

We use this showcase as an opportunity to generalize the simulation for a logistic regression, that we have developed so far, to a version with a flexible number of variables M. This simulation applies to defect modeling the same way.

Listing 5.8: Simulating M variables for N repositories.

```
N <- 120 # Number of repositories.
M <- 4 # Number of variables to examine.

Xs <- matrix(rnorm(N * M), nrow = N, ncol = M) # Producing a N * M matrix of random
    normally distributed values.
```

The previous code produces a matrix of artificial normally distributed variables. It contains N rows for the repositories to examine, and M columns that store M variables for each repository. By adjusting N and M, we can change the characteristics of this simulation.

Next, we need to produce the binary output that reflects the decision for one of the two query languages, based on the variables in matrix *Xs*. In essence, this is the same as previously presented for defects in Listing 5.2. However, we need to change this code to operate on a matrix with a flexible number of variables.

Listing 5.9: Simulating the decision for M variables and N repositories.

```
# Producing a vector of M + 1 random betas, including one variable for a random intercept.
betas <- rnorm(M + 1) # Random betas.

# Adding a column of 'ones' at the left of the matrix, later multiplied with the first beta and
    serves as intercept.
```

```
5 Xs <- cbind(1, Xs)
6
7 # Matrix multiplication ('%*%') of Xs and betas, then applying the logistic function.
8 prob <- 1 / (1 + exp(-(Xs %*% betas))) # The probability deciding for one of the two
      languages.
9
10 # Producing a decision (same as in the previous simulations).
11 Y <- rbinom(N, size = 1, prob = prob)
```

Instead of setting *alpha* and *beta* as individual variables in the code (as done in Listing 5.2), we generalize and use a vector called *betas* (line 2). This vector includes the intercept *alpha* as its first entry. The vector is chosen randomly, since we do not care about the particular effects of variables in the simulation.

As a convenient way to cope with the intercept *alpha*, which is somehow special in not being multiplied with a corresponding variable, the matrix *Xs* is extended by a column of 'ones' on the left (line 5). The matrix multiplication (written $\%*\%$, line 8) then multiplies each variable (including the ones) with the corresponding betas (including the intercept), forms the sum, and thereby produces what is finally feed into the logistic function.

Rating Results

We can now examine the correspondence of betas identified by models using a binomial and the wrong normal output distribution. Furthermore, we can do this on simulated problems with a flexible number of variables and repositories. We will limit us to M and N as set in the previous code. The parameters can be explored systematically using the online resources.

In Figure 5.9, we show six simulation runs that apply the wrong and the correct model. In each gray block, we show a single simulation run, depicting the simulated betas excluding the intercept ($\times$), and the counterparts identified by the logistic ($\bullet$) and linear regression ($\circ$). Simulated and identified variables are scaled to have a standard deviation of 1.0 to make them comparable.

We see that simulated betas ($\times$) are typically different from the identified betas ($\bullet$ and $\circ$). This is not a surprise and reflects the discussion of uncertainty (see Sec. 5.3.2). However, we also notice that the identified betas by both models ($\bullet$ and $\circ$) are close to each other in the six runs, although one of the two models uses a wrong output distribution that does not correspond to the simulation.

Conclusions on Validity

In essence, this shows that we can trust the identified betas, and interpret them, if ignoring their scale. The distribution type of the output is not relevant for this particular statement that we did in our original work. This is an interesting insight as is shows how methods behave if not exactly mirroring the assumptions of the simulation.

However, we want to emphasize that other properties of the wrong model will indeed be wrong. For instance, estimates of uncertainty for the betas will be wrong. We can extend the simulation to show this.

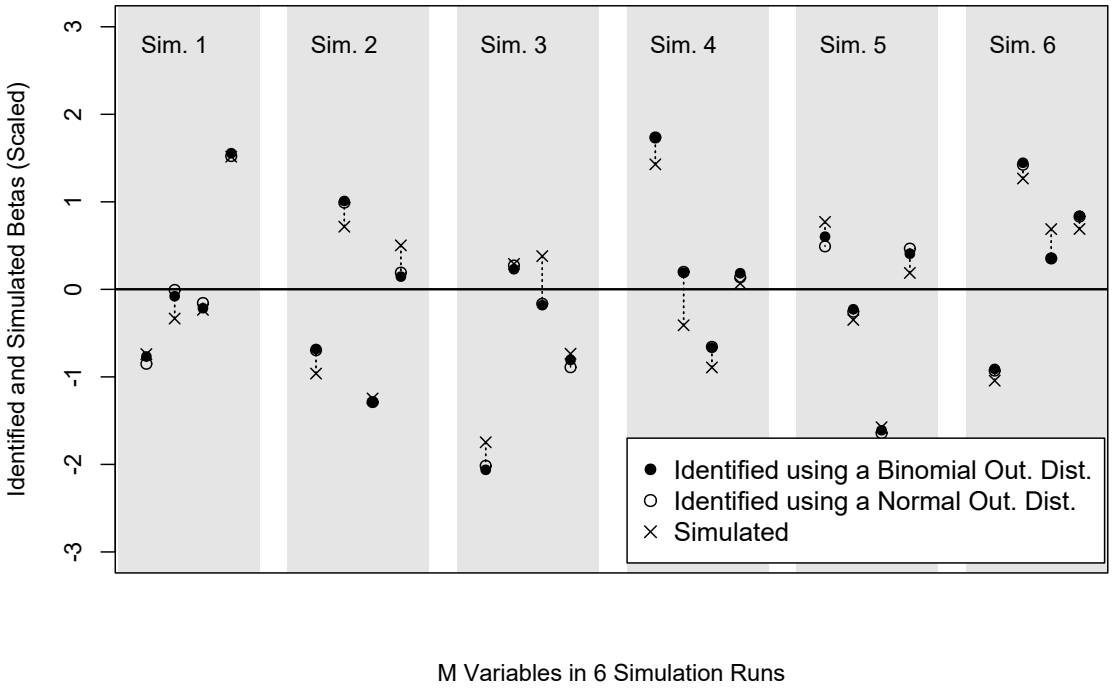


Figure 5.9: Running simulations to explore the difference between logistic and linear regression for the identification of betas.

The insights that we presented here conform to general statistic literature [CCWA13] (page 483). Extracting this cookbook instruction from literature and transferring it to our MSE/ESE cases, when needed, is not easy. We produced the same insights by a simulation-based test here.

5.4.7 Experimental Research (Case 6)

For the last example, we will switch to an experimental method, as an alternative to the analysis of existing data observed from repositories. Experimenting provides major benefits, in particular, in the examination of causal relationships [SCC02]. This is because some assumptions on our scenario can be fixed by the experiment's design. Works on software engineering and ESE research relies on this too [MB20, JLY+19, TLPH95, SHH+05, AC14, AS05, ADGC14, MKPR11, JSD+20, SKP+20].

Research Question

We ask a simplified research question that may be part of many experimental studies in software engineering. This case does not correspond to a specific study but unifies aspects of the studies listed above. We ask a research question, simplified as follows:

- **RQ***: Does our tool improve software development?

We are interested in operationalized statements about the validity of the experimental results. We instantiate our meta research question for the study accordingly:

- **RQ*1**: What assumptions of the study on the design of an experiment for testing a tool can be operationalized?
- **RQ*2**: What is the impact of such assumptions on the result of the study, regarding the improvement of software development by the tool?

In this section, will show two apects on validity:

- We emphasize the strength of experimental research in ESE by operationalized assumptions on the study design. In experimental research, some assumptions are plausible by design, which is an interesting difference to observational studies.
- We show the importance of methods delivering confidence estimates for the interpretation of results, and basic insights of a power analysis.

We believe that even such basic discussion is important because experiments are widespread (see instances like [AC14, AS05, ADGC14, MKPR11, JSD+20, SKP+20]). However, they are also run by researchers that encounter this kind of practice for the first time. The simulation of an experiment can help to raise a researcher's confidence in a correctly applied method and corresponding statistic devices, upfront to the actual execution of the experiment.

Original Method

We stick to a basic experimental design to prove the benefit of a new tool. The method that we follow can be considered as a simplified version of instances found in related work, e.g., examining the effect of artifact formats [AC14], the role of use cases [AS05], unit testing techniques [ADGC14], performance evaluation of software architectures [MKPR11], textual vs. graphical software design descriptions [JSD+20], or run-time configuration frameworks [SKP+20].

We are doing a randomized experiment with 20 subjects. We start with the selection of the 20 subjects. Ideally, the subjects conducting our experiment are representative of a population. What this means is hard to formalize, and best described by the process on how the subjects are selected. We will ignore aspects of sampling.

The experimenters then randomly assign each subject to the *treatment* or *control* group. A subject may use our new tool as treatment, or an established (or no) tool as control. We measure the outcome of the experiment in terms of the time each subject needs. Depending on the group, this may be with or without the new tool. Finally, we compute the difference in time, needed by subjects in the treatment and control group.

Simulated Scenario H

In the following simulation, we simulate the experiment producing an artificial version, where our new tool improves how a subject handles a task. The time needed is reduced by four minutes.

We will hard-code relevant parameters, like the number of subjects (20), the effect of the tool (- 4 minutes), or the severity of difference in the subject's preconditions (given by a standard deviation of 5 minutes). We recommend changing these parameters and exploring the implications based on the online material to get a better understanding of the impact on results.

Listing 5.10: Simulating a randomized experiment.

```
1 N <- 20 # Number of subjects.
2 S <- rnorm(N, mean = 60, sd = 5) # Unobserved preconditions of subjects.
3
4 # Randomly assigning treatment and control.
5 G <- sample(c("treatment", "control"), N, replace = T)
6
7 # The effect of our treatment.
8 X <- ifelse(G == "treatment", -4, 0)
9
10 # Composing the time that a subject actually needs.
11 Y <- S + X
```

In a first step, the code decides on the number of subjects N (line 1). We simulate the preconditions of subjects as the unobserved variable S, i.e., the time a subject would need for the task ignoring tool support. We use a stochastic function following a normal distribution with a mean of 60 minutes and a standard deviation of 5 minutes (line 2). We thereby have artificial subjects as unobserved preconditions S.

Following the standard method of randomized experiments, we now randomly assign our subjects to treatment and control group. We use the stochastic function *sample* to do such random assignment to *treatment* or *control* (line 5). We store it as observed variable G. We then simulate the effect of our new tool X for the assignments, as a function of G, using a basic *ifelse* (line 8). We make our treatment (new tool) decrease the time needed by exactly four minutes.

Here we face an important difference to our previous definitions of X, e.g., in Sec. 5.4.3, since we can assure by the design of the experiment, that G, and thereby also X, is independent of anything else.

Finally, we simulate running the experiment, producing the time Y as the sum of X and S (line 11). The time is decreased by our treatment but still influenced by each subject's unobserved precondition.

Rating Results

Obviously, we should recognize that the experiment is a success by just looking at the difference between treatment and control group's average time needed for the task. Our simulation clearly defines that the treatment (the tool) decreases the time by four minutes.

However, the simulated random preconditions of subjects will complicate showing the success of our tool. We can illustrate the invalidity of an over-simplistic method that only interprets the difference between treatment and control group, by repeating the simulated experiment many times. We record plain difference between treatment and control (see online resources) and report on it in the histogram in Figure 5.10.

Fortunately, most of the simulated experiments suggest that our new tool is

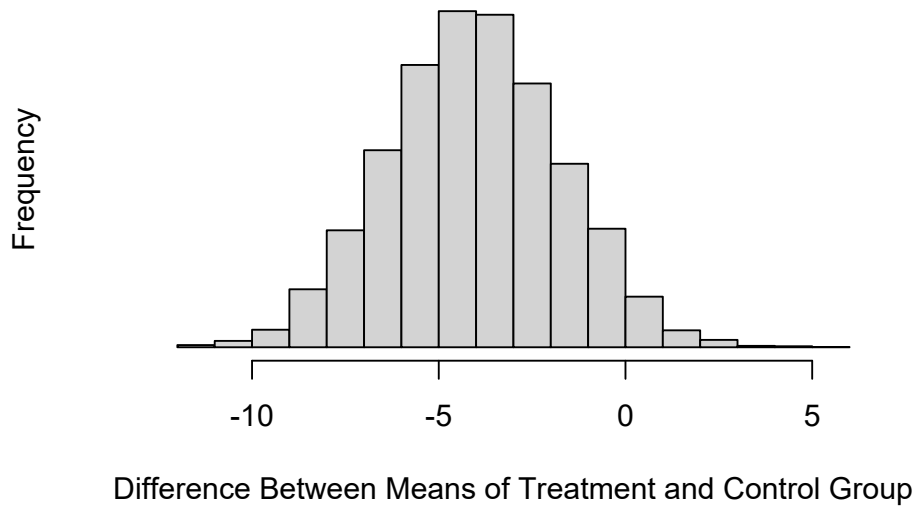


Figure 5.10: Repeating the simulated experiment and interpreting the plain difference between treatment and control group.

indeed a success. Often, we almost exactly meet the −4 minutes. However, there are also cases where we run into simulated experiments that suggest that the new tool slows down the task. We know that this is not the case, since the simulation is fully transparent.

The reason behind these cases are unlucky assignments of subjects to treatment and control, where the majority of the subjects with good preconditions concentrate on the control group. This is something that happens. There is no way to resolve this problem, without a better observation of the preconditions (which is often impossible).

Revision

We will now show the conceptual remedy, that resolves this problem by reasoning about the uncertainty associated with the preconditions of the subjects. It is the typical device of a t-test or confidence interval, that almost all studies doing experiments are aware of. We again focus on confidence intervals, which we believe are more intuitive for most readers.

Instead of interpreting the plain difference that we find between treatment and control group, we now prefer to interpret a confidence interval around this difference. We illustrate this in Figure 5.11 in the same way as we have done it previously in Sec. 5.4.2. The confidence intervals for 10 simulation runs uncover two important insights:

- First, we resolve the threat of accidentally claiming a wrong effect of our tool. In all simulation runs, we see that the confidence interval includes the simulated effect of our tool −4. When running the actual experiment, we can thereby be sure that that our statement, which is now less accurate by including a notion of uncertainty, is valid in claiming the success of our tool under the simulated assumptions.

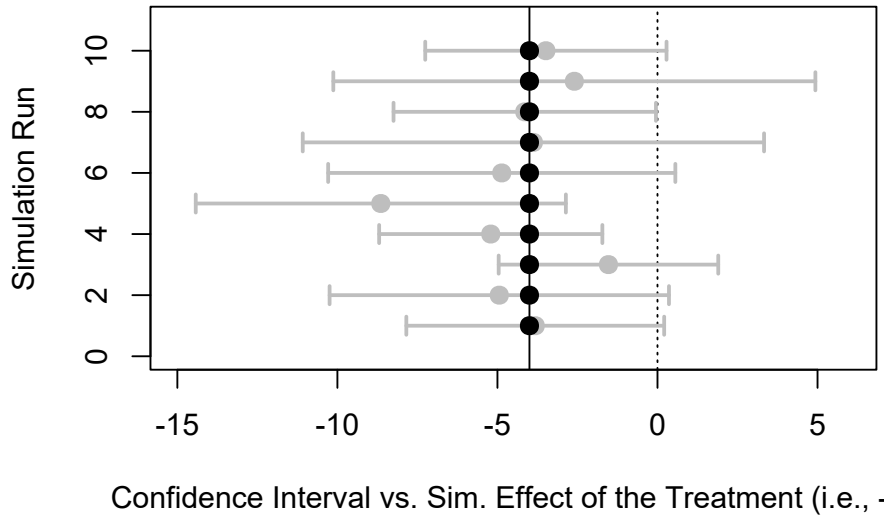


Figure 5.11: Showing 10 simulated experiments and the corresponding uncertainty for the difference between treatment and control. The black dot reflects the simulated difference, the gray dot the difference that we find, and the gray bar the 95% confidence interval around the difference.

- However, we spotted a new problem. Most of our confidence intervals (8 out of 10) suggest that we cannot be sure that our tool has an influence at all because the 0 is included in the intervals. Some people say that we missed rejecting the null hypothesis. This operationalizes invalidity of statements about the '*absence of an improvement by our new tool*'. The simulation states that the effect is there. If we claim absence because of a confidence interval that included the 0, we are wrong. This may happen for about 80% of the runs, but we can decrease the chance. For instance, by increasing the number of subjects, the confidence intervals will get more narrow and the 80% will decrease. We do this until we think the chance is small enough to be accepted. If we now still face a confidence interval including the 0, the effect of our tool may be truly negligible. We face an instance of a power analysis based on a simulation.

Conclusions on Validity

The important property of such experiment is that the *independence* of the assignment to treatment and control can be assured by the design of the experiment. We operationalize this assumption in our simulation.

As the consequence, this assures the *independence* of the effect of the tool used in the experiment (the effect that we are actually interested in). We can continue to reason about causation, as sketched in Sec. 5.4.3, while having the most relevant assumption in the simulation plausibly being assured by our design. There is no need to control for variables, as required in Sec. 5.4.4.

Furthermore, this example again revisits aspects of uncertainty, first presented in Sec. 5.3.2, and refined in the context of dependent observations in Sec. 5.4.2. However, having dependent observations in experimenting is not completely im-

plausible. Subjects might influence each other when running the experiment in the same room, or we might design the experiment so that we measure the same subject executing multiple tasks. Such dependent observations require advanced methods, e.g., subject-item designs, examined in simulations in [BDB08].

We also distinguished between claiming the existence and non-existence of effects. We show a basic version of a power analysis by a simulation. We recommend exploring the online material, adjusting N, the effect of the tool, or the severity of difference in the subject's preconditions given by the standard deviation for simulating S, to examine the impact for resulting claims on existence and non-existence.

Chapter 6

Conclusion

This paper describes and validates a new strategy to validate methods and results of empirical research studies. While reproducibility and replicability are somewhat understood, standardized and operational ways to define and communicate such validity of empirical research studies are less understood.

Our strategy operationalizes important assumptions, that are typically informal in papers, by simulations. We use the simulations to show how the assumptions impact the results of a study. We call the strategy *simulation-based testing* because of the analogy between writing simulation artifacts and test cases.

In a (meta) validation, we show how simulation-based testing instantiates research questions on the validity of studies in six real scenarios. We show that we can either support validity, threaten validity, or invalidate studies.

We encourage researchers to accompany submissions of research works with simulation artifacts, thereby proving the status of an operationalized validation; thereby, helping reviewers in assessing validity. In this way, the reviewing process of future empirical work would be improved.

This conclusion finalizes the last contribution of this thesis. We will now continue with a summary of the limitations of our methods, a collective discussion of the related work, and finally draw an overall conclusion.

Chapter 7

Limitations

This chapter summarizes the limitations of all three contributions included in this thesis (Chapter 3, 4, and 5).

For the incremental map-reduce and Datalog method, the technical limitations are mostly inherited from the corresponding general usage of map-reduce and Datalog. Most limitations of map-reduce and Datalog are well-known. There are some additional details discussed in this chapter, caused by tailoring such methods to the computation of abstraction of repositories (potentially focusing on a heterogeneous technology stack, or incrementalization).

For the simulation-based testing high-order method, the limitations are not obvious. There are no real limitations on how to write simulations, but there may be limitations in the plausibility of certain simulated scenarios. Plausibility needs to be discussed as parts of concrete research. In the following, we will refine this idea.

7.1 Incremental Map-Reduce on Repository History (Chapter 3)

Technical computations using our incremental map-reduce prototype are limited in close analogy to regular map-reduce frameworks. The general framework of map-reduce has previously been used in the context of MSR/ESE, which is an indication for the suitability [SJAH09, SAH10].

The application of map-reduce in MSR/ESE is somewhat special, as computation involves accessing repositories and its fragments. Our method is special because it scales such computation by mechanisms coming from incrementalization theory.

7.1.1 Core Interface

The core interface of our prototype is not limited to map-reduce, but in terms of constraints on the data types, that need to have an algebraic structure for the change representation, and in terms of functions that need to have (self-maintained) derivatives.

While such abstract limitation of the core interface is interesting for the concise realization of our prototype, concrete limitation for a user of our prototype comes with the map-reduce framework that we build on top of the core interface. A

typical user will not need to program against the core interface or provide specialized algebraic structures.

The data type processed by our method must have a conforming algebraic structure. Since we aim at implementing map-reduce on top of the core interface, we need a collection type. We provide an algebraic structure (Abelian group) for bags by default, so a user does not need to get in touch with the change representation. Bags are containers that are flexible enough to store all kinds of abstractions of the source code, no matter of what data type. Such a bag may also hold key-value tuples to capture the structure of the data, e.g., to maintain a connection between a class and a metric. Set semantics of the collection can be realized by a function computing the distinct elements. List semantics is complicated to incrementalize, so we exclude it. There is always the option to escape incrementalization, and to fall back to standard map-reduce without acceleration.

To get an incremental performance benefit over regular map-reduce, the core interface limits a function to have a proper (self-maintained) derivative. For our placeholder, we have just been illustrating how to compute and aggregate cyclomatic complexity, but our method allows incrementalizing the full range of extended map-reduce primitives, including map, filter, group-wise aggregation, joining and the Cartesian product. Such functions operate on bags, possible with key-value entries. More complex processing is limited to chaining these functions, analogue to a conventional map-reduce framework.

7.1.2 Pure Functions

The high-order `map` function is one of the most important when it comes to discussing limitations. The `map` function takes another function as a parameter and applies it to the elements of the bag. This can be used to plug a wide range of analysis code and apply it to the individual resources.

The `map` function has an important but somehow obvious limitation. The function which is passed as a parameter needs to be pure so that we can guarantee that the self-maintained derivative of `map` is pure too. Otherwise, our method will fail. However, implications of this limitation in the context of repository revisions are not immediately obvious. We illustrate this using a basic example.

Consider the following two Scala objects, shown in Listing 7.1 and 7.2, contained in different resources of the revisions (e.g., in the files 'Program.scala' and 'Library.scala'). We want to associate parts of the program with types.

```
1 object Program {
2     val out = Library.compute()
3 }
```

Listing 7.1: A Scala program

```
1 object Library {
2     def compute(): String = "Some String"
3 }
```

Listing 7.2: A library object the Scala program uses

Naive type resolution would use `map` to apply a function to the individual resources. We may reuse an existing solution to parsing and type resolution. However,

the type resolution in this example depends on the context. If we try to resolve the type of variable `out`, we need the Library file to do so. Here we face a problem since the Library file may change over the revision history too (e.g., the return type of `compute` may change). As a consequence, the type resolution function, applied to a single resource, will not be pure.

Such cascades of changes renders incremental type resolution, done over multiple revisions, to be a complicated problem. Type resolution is definitely a very elementary problem to every code analysis, but even closely related work, like LISA [APPG19], does not discuss it in-depth (or solve it). They also use `map` for individual resources.

We assume that work on incremental static program analysis can contribute ideas for future work on this limitation, e.g., [Sza21, SBEV18, dPSER20].

7.1.3 Recursion

The missing support and evaluation for recursion or fixpoint operations is another limitation of our method. We assume that we can solve recursive problems, using standard iterative methods that chain map-reduce primitives until a fixpoint is reached. This is comparable to the usage of map-reduce in, e.g., [MAB+10]. However, we did not conduct any experiments on this option, so we still consider it as a limitation.

7.1.4 Abstracting Across Individual Revisions

We only contribute static abstraction of single revisions, computed for all revisions of the repository. This contribution may be an important building block for more advanced abstractions of the full repository history. However, functions that abstract over more than a single revision are currently excluded from our method. We discussed this in the introduction of Chapter 3 and suggest some ideas for resolving this in the background section of the same chapter.

7.1.5 Usability of Map-Reduce

Another question is if we can assume map-reduce to be used by software engineering practitioners, educators, and researchers to express MSR-like queries. The contribution done in this thesis focuses on the technical aspects of incrementalizing map-reduce with the aim to improve scalability when abstracting over repositories.

We do not study the limitations of usability for map-reduce in different user groups. This is beyond the scope of this thesis. However, map-reduce is popular and has already been used for MSR/ESE before [SAH10]. These are strong indications that map-reduce can be used by a wider range of user groups.

7.2 Repository Mining with Datalog (Chapter 4)

Technical computations using our Datalog method are limited in close analogy to regular Datalog. There are some differences because we aim to transfer the ap-

plication, from understanding a homogeneous, to understanding a heterogeneity technology stack in a repository.

7.2.1 Limitations of Datalog

The limitations of Datalog are well-known [GHLZ13]. Related work in architecture recovery [MMW02, MT01, TM03], source code querying [HVdM06], and static program analysis [BS09, SBEV18] shows that these limitations do not prevent computing advanced abstractions of the source code and its surrounding. Potential limitations may be relaxed by adding features to Datalog, or using closely related declarative logic programming languages, often being a compromise between limitations, assertions and efficient algorithms.

7.2.2 Accessing the Repository by Pure Functions

The method of using pure functions to access the repository and its fragments is very flexible. We may plug arbitrary (high-order) functions into such mechanism and drive access selectively by Datalog rules. There should be no access that cannot be decoded as a chained application of functions. We provide an initial catalog of high-order functions for accessing standard formats, which may need to be extended to also cover other subject technologies.

A limitation with such a method is that different ways of accessing the same fragment leads to difference entities in the Datalog reasoning. Here the modularity of our method ends, and users writing rules need to agree or normalize access to the content. Aligning the same entities in different knowledge representations is a well-known problem [ZLH+21]. It does not fit into the scope of this thesis.

7.2.3 Usability of Datalog

Like previously discussed for map-reduce, it is not obvious if our Datalog method can be used by software engineering practitioners, educators, and researchers to express MSR-like queries. The contribution done in this thesis focuses on the technical aspects of such solution, but also provides a concrete case study that shows that such method is suitable.

Our study can be seen as a reproduction of previously reoccurring ideas on architecture recovery [MMW02, MT01, TM03], source code querying [HVdM06], and static program analysis [BS09, SBEV18], evaluated in a novel application context. The continuously reoccurring idea of declarative logic programming to solve related problems is a strong sight for the usability.

We also assume that different use groups can benefit from it, since previous work, like [SB10], point out the ease of configuring such solutions. Future work may combine such efforts and come up with ‘copy and paste’ solutions (or even AI-based solutions), composing mining logic with minimal manual intervention.

7.3 Simulation-Based Testing (Chapter 5)

For the simulation-based testing high-order method, the limitations are not obvious.

7.3.1 Technical Limitation

There are no real limitations on how to write simulations. We have shown how to write simulation in R, but such code can be transferred to any other programming language. There are some features that facilitate writing simulations, like built-in random number generators, vectorized operations, and matrix libraries.

We also assume that there is no 'correct way' of designing simulations. Orthogonal approaches, based on actors, may also contribute plausible ways of simulating aspects of the software development.

7.3.2 Plausibility Limitation

We may be limited in the plausibility of simulated scenarios. Simulations are based on assumptions, that may or may not be true. For the way we use simulations to test methodology (methods and models), the fact that an assumption is true, does not immediately matter. We may, for instance, show that a methodology works independent of an assumption. Often this is not immediate obvious in complex scenarios.

However, in some cases, a methodology may fail for a given assumption. This can render a threat to a methodology. The simulations only disambiguates the threat and its impact. If the assumption is true, cannot be rated by our high-order method. Plausibility of assumptions is subjective and needs to be discussed as parts of concrete research.

7.3.3 Conclusions on Reality based on Simulated Data

Methodology, methods, models, and simulations are very related in that they are collections of assumptions and everything that can formally/operationally be derived. They are useful to derive results from the assumptions that are not immediately obvious to us.

For instance, we might have assumptions on relations between variables, assumptions that some parameters exist, assumptions on the meaning of data being our variables, or even assumptions on the process on how to connect all this best. All assumptions are subjective (or axiomatic), but we have already agreed on some of them, mostly because they have been shown useful.

Take the statistical methods, as an instance, that help us to reason about how reality behaves under clear assumptions. On the contrary, our idea of a simulation helps us to reason about how statistical methods behave under clear assumptions. We consider our use of simulations as some sort of high-order method, further relaxing, changing, or strengthening various assumptions systematically, to facilitate our understanding of an underlying set of methods. Seeing assumptions at work, instead of just having them fixed, or knowing them from discussions, is often helpful.

An irrational expectation of the use of simulated data is that it can help to partially or fully replace real data in its meaning to understand real phenomenon. A good example is the intuition that replacing privacy critical data by synthetic data may help. If we start working with assumptions instead of real data, we may argue, but typically, we move away from reality.

Chapter 8

Related Work

This chapter presents the related work, divided into the three contribution types that we have previously introduced in Chapter 2. We will start with technical contributions to methods, followed by methodological contributions to high-order methods (mostly related to simulations), and finally list some concrete empirical contributions that might benefit from such (high-order) methods.

We exhaustively discuss work positioned in MSR/ESE. However, this section will selectively discuss work coming from different fields that shares a strong relation to our contributions. For empirical contributions in Sec. 8.3, we will limit us to the field of ESE/MSR.

> To improve readability, we highlight the passages discussing the relation between the related work and our contributions (comparable to this paragraph).

8.1 Technical Contributions

The two technical methods we contribute are: i) incremental map-reduce to scale the computation of abstractions of repositories, and ii) Datalog to abstract over repositories with a heterogeneous technology stack. We examine the related work following this structure:

- The top-level structure iterates through principle decisions for the proposed methods, starting with the used languages and interfaces, ways to scale computation, options for data storage, and special properties of the data under study.
- The second-level structure distinguishes between fields, e.g., between very general, and MSR/ESE-specific work. The related work discussion is exhaustive for MSR/ESE. Other fields are not exhaustively discussed.

8.1.1 Languages and Interfaces

A language or interface in which a data analysis can be written is one of the key properties of broadly applicable data analysis methods in MSR/ESE. Decisions have an impact on following up options, e.g., for the scalable computation or storage.

General In the following, we will discuss general methods to data analysis, going from functional, to declarative logic, to imperative.

> The following general methods are not designed for, or evaluated at, the analysis of repositories and its fragments (in the field of MSR/ESE). Our two technical contributions built up on such general work. However, we tailor and evaluate them for the analysis of repositories (potentially with a heterogeneous technology stack).

Map-reduce is one of the popular functional interfaces for the processing of data [LLC+11]. There may be a distinction between basic and extended map-reduce, where the latter also allows operations like join or the Cartesian product. Solving recursive problems in map-reduce can be done by iterative solutions [ELZ+10]. Large-scale graph computations, based on map-reduce, are closely related and can be expressed in Pregel [MAB+10].

Relational algebra like interfaces and languages, like SQL, Pandas DataFrames, R DataFrames, or Spark DataFrames, are other functional option to data analysis. SQL, for instance, is used in MSR/ESE, like in [RvDV13, Gou13]. Such methods are related to map-reduce in the operations they provide (e.g., join, map, or filter), but relational algebra in more specific in focusing on tabular data. Focusing on tables allows some automated optimization of data analysis, e.g., by rewriting queries [AXL+15].

> Our incremental map-reduce solution is motivated by such general work on map-reduce and relational algebra. We provide a map-reduce interface that allows users to compose processing. We do not provide dedicated support to work with tables, but users can process bags of tuples by map-reduce operations.

Datalog [HGL11, GHLZ13] and Signal/Collect [SBC10] are declarative, since an analysis can be written independent of a sequential order. Such specifications can be composed out of independent modules. Datalog builds on top of relational algebra to store facts, but infers new facts using a set of rules. Signal/Collect builds on top of a graph, sending and consuming messages along the edges, to infer facts at the nodes of the graph. Datalog and Signal/Collect allow recursion natively.

> Our Datalog method is motivated by such general work on Datalog. Our Datalog method is different from Signal/Collect.

Previous options may work with streams of data too, which requires the underlying execution engines to work on changing data. One example is Structured Streaming for Spark, that provides functionality close to Spark DataFrames, while reacting to updated data [ADT+18]. Such options involve data flow abstractions, such as sources, sinks, and stores [BROL14, ADT+18].

> Our incremental map-reduce method is close to such notion of changing data and the underlying incremental update. However, we tailor our method towards processing the repository history as the primary data source.

Data analysis may always be written in general purpose imperative programming languages, such as Java or C# (e.g., see the analysis in [SPN+18]).

Architecture Recovery In the field of architecture recover, Mens et al., Mens et al., and Tourwe et al. (see [MT01, MMW02, TM03]) focus on the usage of declarative logic programming to recover different aspects of the software architecture, like design patterns or programming patterns. Such data is used, for instance, to provide support during refactoring activities.

To this end, the authors use declarative logic meta-programming, i.e., a declarative logic metalanguage to operate on, and formulated queries to, an object-oriented base language. As metalanguage, the authors rely on a Prolog-variant. Authors experiment with Smalltalk and Java as base language. To connect the metalanguage with the base language, an interface is introduced that describes the representational mapping between both.

> Our method using Datalog can be considered as a recent reproduction of such methods in a different context. The logic rules that we use for inference are almost similar to what the previous authors use. We differ in tailoring our language to examine a more heterogeneous technology stack, not limiting our analysis to a well-defined AST of a single base language. We consider EMF as a heterogeneous technology to be subject to analysis, involving different kinds of artifacts, including the build system, Java, OSGI, or XML. Compared to the previous representational mapping interface, our accessor primitives are designed to be more flexibly to bridge such gap. Our access to a repository starts at its root. Hereafter, we can navigate the fragments by chaining arbitrary high-order function calls. During such traversal, we might potentially access, and decompose, different artifacts types, building abstractions, like ASTs, on the fly. We always have a default identification of the fragments of the repository by the uninterpreted function terms used to access them.

Source Code Querying In the field of source code querying, Hajiyev et al. [HVdM06] propose to use Datalog. The aim it to query Java source code using Datalog as part of an Eclipse plugin. Access to the AST is granted over an indirection over a database. The authors also do initial experiments on incrementally updated queries.

> The rules used by the previous authors are almost similar to the rules in our Datalog method. We see this as a sign of the applicability of Datalog to query source code and related artifacts. However, our method differs in allowing a more flexible access to the heterogeneous content, not just being tailored to a single base language.

In [RNKJ11], the logic program query language SOUL is used to query Java code that is part of Eclipse JDT projects.

> We focus on a heterogeneous set of languages involved in the technology stack subject to analysis.

Static Program Analysis Work in the context of static program analysis picks up concepts from declarative logic programming. In [EKKM08], program dependencies are analyzed using Datalog. In [BS09, SB10], points-to-analysis is implemented

using Datalog. Authors come to the conclusions that: '*implementation is modular and can be easily configured to analyses with a wide range of characteristics, largely due to its declarativeness*' (direct citation [SB10]).

> The previous methods on static program analysis show rules almost similar to our rules. The representational mapping is different as it is tailored to a homogeneous base language. The conclusion that definitions are modular and easy to be configured supports the modularity of our method. Advanced static program analysis may be a reusable component that we might add to our inference in future work.

MSR/ESE We now discuss work in the MSR/ESE field. The DJ-Rex solution [SJAH09, SAH10] motivates the migration of existing analysis to map-reduce. Boa [DNRN15, DNRN13, NDNR14] uses a DSL for visiting the Java ASTs in a distributed manner which is backed by distributed map-reduce.

> We have previously discussed both solution in Chapter 3. Analysis code in Topleet and DJ-Rex are comparable. The difference comes with the performance. For the usability, relying on established map-reduce may also be a benefit, compared to introducing a novel DSL.

LISA [APG17, AG15, APPG19] enables the usage of Signal/Collect [SBC10] to process the history of an AST created by plugging a parser. As previously discussed for Signal/Collect, LISA allows a declarative way of formulating queries by specifying messaging. Messaging is done along the edges of the AST.

> Such method using Signal/Collect is different from map-reduce, but closely related to our Datalog method. However, limiting messaging to a previously defined AST, as LISA does, cause difficulties, because of lacking an option to add edges/nodes during analysis. This might be necessary to reason about implicit connections in the AST, e.g., like needed for points-to-analyses or type resolution.

In work by Stevens et al. (see [Ste15, SRN+14]), declarative logic programming is aligned with a graph query language to abstract over the revision history. The representational mapping allows accessing to source code of all revisions and thereby enables computing abstractions over the full revision history. Regular path expressions are used for querying such revision graphs.

> The presented solution provides abstractions over the repository revision history, like our map-reduce method, but it uses declarative logic programming, like our Datalog method. However, we focus on incrementalization and on a heterogeneous technology stack.

There are very basic methods to access the repository data. We may use the Git command line [SG18], or Git wrappers, like JGit[1] or PyDriller [SAB18]. In [LDKBJ22], specific processing, like reference resolution, are optimized by a multi-revision AST. Authors of [GCB+21] and [HHK20] limit their work to the history of source code methods.

[1] `https://www.eclipse.org/jgit/`

> Such work does not propose methods to write custom analysis. There is no elaborate language or interfaces to compose custom abstractions of repositories and its fragments. We examine map-reduce and Datalog to produce such abstractions.

8.1.2 Scalable Computation

A second aspect of data analysis is how computation is scaled. Solution can scale at the price of adding computation resources; other improve performance by memorizing intermediate results or applying incrementalization.

General Map-reduce frameworks provide distributed and parallel computation almost by default [LLC+11]. Pregel [MAB+10] is optimized with respect to iterative graph computations, but comparable to map-reduce in its distribution. Signal/Collect [SBC10] shares a stronger relation to actor frameworks facilitating asynchronous messaging. Actors systems can be distributed [Agh90]. Stream processing can be distributed [CBB+03]. The processing of relations can be distributed [AXL+15]. Datalog queries can be run in parallel and in the cloud [GST90, ACC+10]. All such solution scale at the prices of adding computation resources.

> Our methods inherit such capabilities from the general methods. Our map-reduce method allows distributed and parallel processing. The Datalog method does not yes employ comparable mechanisms, but conceptually they are applicable.

Solutions to incrementalization, memoization and redundancy reduction are somewhat related. All solutions split the data analysis into sub-problems, and thereby facilitate reusing existing solutions to make analysis scale. Methods do not necessary scale at the expense of additional computation resources, which is a benefit because it saves energy.

Most of the general methods provide extensions to existing analysis languages and interfaces, to realize incrementalization, memoization and redundancy reduction. Such strategies are employed in the batch processing platform Dryad-Inc [PBYI09], reusing identical computations and processing changes. Closely related works are [LOR+10, YYY+12, BWR+11]. Systems sharing a corresponding understanding of abstract algebra, such as monoid structures and monoid homomorphisms, are [HS13, BROL14, Feg16]. The systems described in [BROL14, Feg16, ADT+18] concern incremental stream processing. In the database literature, incrementalization occurs as view maintenance [CY12, QGMW96, GMS93]; production systems are incrementalized in [For82]. The usage of Abelian groups for incrementalization can be found in [GGMS97] for view maintenance, and in [CGRO14] for incrementalization by program transformation. In [LW97], incremental aggregation is discussed. All such solution do not scale at the prices of adding additional hardware.

> Our efforts on incremental map-reduce is motivated by such general work, but specific, since it aligns incrementalization with the revision history. Our Datalog solutions does not systematically explore such optimizations yet. Conceptually, Datalog can be incrementalized.

MSR/ESE DJ-Rex and Boa are both executed on a distributed map-reduce framework; LISA applies local parallelization, not employing the option to distribute Signal/Collect [SBC10].

> Topleet delegates its core operations to a distributed map-reduce framework and thereby inherits distribution. Self-maintained derivatives are efficient to distributed, regular derivatives may require shuffling of data. Our Datalog method does not provide any support for distribution or parallelization yet, but such improvement is conceptually suitable, as described in [GST90, ACC+10]. It may be part of future work.

LISA covers the reduction of redundancies. It tries to merge similar AST nodes and messaging as much as possible, employing the similarity of succeeding revisions. The underlying Signal/Collect infrastructure enables recursion.

> LISA does not build on theoretical foundations, like abstract algebra and derivatives, which makes the proposed optimizations seem more improvised. LISA filters and maps the changed resources during initializing the AST; both operations conform to self-maintained derivatives (homomorphisms). LISA does not allow using them in different ways, or to plug custom derivatives. In our map-reduce method, we allow the usage of extended map-reduce operations where all functions have known derivatives. We also allow plugging custom derivatives. However, LISA natively allows recursion, which our map-reduce method cannot provide. Our evaluate show that we outperform LISA in terms of time and memory, on a simple task computing the cyclomatic completely. However, the capabilities of map-reduce and Signal/Collect still strongly differ, making such comparison only partially meaningful.

Boa and DJ-Rex do not employ related concepts to improve scalable computation without adding computation resources. They scale at the price of additional computational resources.

> Currently, there is no other method that enables computing static abstraction of repositories accelerated by incrementalization. Our Datalog solution does not focus on scalability.

8.1.3 Storage

Query mechanisms, like the ones we propose, should typically not be designed without thinking about mechanisms to store data.

> Optimization of the underlying storage is not the central research aim of our technical contributions. However, we list some important related work and options for future work.

General There is a wide range of applicable techniques for efficient data storage. When using distributed map-reduce, in particular Apache Spark, there is a range of alternative serializers (like Kryo) or tabular data formats (like Apache Parquet [Voh16]). Other options include databases, like standard SQL, or Key-Values stores, to persist data.

> We rely on the storage mechanisms of the underlying frameworks we build our solutions on. We do not examine alternative storage mechanisms.

MSR/ESE While there is a dedicated category for work in MSR/ESE providing concrete data sets (e.g., [NBKO21, VSF15]), the more general challenges of data storage in MSR/ESE can be understood as the practice of scalable archiving of software engineering data, and abstractions thereof. Such work needs to efficiently protect against loosing data permanently, that eventually turns out to be relevant to future research in software engineering.

Archiving is done by *Software Heritage*, and presented in [CZ17]. Such mechanisms serve as an important alternative to conducting empirical research on proprietary GitHub. Other efforts, like GHTorrent [GS12], recently stopped active archiving. The latter case shows the importance of initiatives to archiving.

> Our work is limited to very basic data storage. We allow access to a Git repository, but we may be able to adjust our interfaces to also access different kinds of archives. We assume that data storage motivates future work that tightly aligns storage and computation.

Authors of [BPVZ20] present counteractions to growing data, i.e., to improve the scalable storage of repository data, by applying graph compression. Compression may be a compromise between computation time and storage space. Such counteractions may not necessarily alight with scalable mechanisms to compute abstractions on the data. We assume that bringing together scalable computation of abstractions and archiving, is one of the important steps for future work on the technical side of ESE/MSR.

> Our incremental map-reduce method completely inherits the storage mechanisms provided by Apache Spark. For our Datalog method, we store the results in a convenient, but inefficient XML dialect.

8.1.4 Structure in Time and Space

Methods, analyzing data, share a corresponding understanding of how the data is structured in time and space. Such notion may be a limitation for the processing.

General Online methods, like those concerned with stream processing [Feg16], focus on low-latency by processing arriving changes. Offline processing, such as [PBYI09, LOR+10, YYY+12, BWR+11], employ the evolution of fully available data (batch) to decrease computation costs. Some hybrid methods switch between both [BROL14].

> All such methods consider temporal evolution as something linear without branching. Our map-reduce method is different as it considers the acyclic revision history.

MSR/ESE Online processing of streams has been discussed in [GSV16] regarding real-time capabilities, adequate query models and data summarization techniques. A stream-based method is GHTorent [Gou13], gathering meta-data from GitHub's push API. Acyclic evolution, i.e., branching, is somehow special to MSR. Formalizing patches, merges and conflicts has been done in [AMLH14, MG13]. Such works provide formalization that does not offer ways to compute abstractions over repositories. MSR methods like DJ-Rex, LISA and Boa can be considered as offline. DJ-Rex and Boa do not make any strong assumptions on the underlying notion of time, since they do not employ it natively to improve scalability; LISA is limited to a linear history. [LDKBJ22] reuses AST nodes along the time and space. The latter refers to reusing nodes part of the same revision.

> Our work is the first that provides optimized (offline) processing of data evolving according to an acyclic repository history, i.e., involving a notion of branching, by incrementalization. Our Datalog analysis does not yet work on multiple revisions.

8.2 Methodological Contributions

This section discusses general work that proposes (high-order) methods or discussions to improving data analysis methodology. We distinguish between fields, starting with statistic science, continuing with work specialized towards computer science, and finally discussing MSR/ESE-specific insights.

Statistic Science There are numerous textbooks that guide statistic data analysis, such as [CCWA13, DB18, Har15, McE20]. Such work applies to several domains. Application examples are often limited to medicine, psychology, or ecology. Such work cites simulation studies as primary source for the justifications of methods, next to mathematical derivations. Mathematical derivations and simulations can be considered as high-order methods to examine, understand and teach statistic methods.

In [Har15, McE20, GHV20], original simulations are developed to show central properties of the presented data analysis methods. We add a direct citation of the first three sentences of the preface of [GHV20] to emphasize this trend: *"Existing textbooks on regression typically have some mix of cookbook instruction and mathematical derivation. We wrote this book because we saw a new way forward, focusing on understanding regression models, applying them to real problems, and using simulations with fake data to understand how the models are fit."* (Direct citation of Gelman et al. [GHV20]). This excerpt makes clear that simulations are used on a different conceptual level, compared to fitting a model (a typical method in the statistic science). They are used to understand the typical method. This is the reason we refer to them as a high-order method.

In [McE20], simulations are used as a central device for the presentation. In such work, simulations can be considered as a lightweight alternative to more formal derivations. As a counterexample, a focus on mathematical derivations used to understand data analysis methods is taking in [DB18].

Statistic work examines cross-validation in a simulation study in [Sha93]. In [RBC+17], cross-validation is evaluated on simulated structural data in the field of ecology. In [BLST13], the impact of random effect structures is examined by simulation. The events-per-variable (EPV) are examined in the context of logistic regression in medicine [PCK+96]. In [GAB+20], the authors simulate what happens if something informative is ignored, which is part of longitudinal health data. The authors of [BARH06] discuss the role of simulation studies in medicine. The evaluation of statistic methods by simulation is discussed by [MWC19] (also in medicine).

> Our work on simulation-based testing is motivated by such work coming from the statistical science.

General Computer Science Compute science adapts many such works from the statistic science, employing the rapid amount of available data and new computational possibilities. Evolving disciplines are data mining, pattern recognition or machine learning [Agg15, JM15, Bis06].

According to our best knowledge, authors mostly provide mathematical derivations, or compare methods working on real data. Authors do not yet rely on simulations to examine methods.

There are many direct adaptations of statistic practice in the context of computer science. In [AB11], guidelines for statistic tests are listed, focusing on algorithms with random components. Such guidelines are collections of references to general statistic science.

Most recently, testing machine learning start to become of interest. A review is provided by [BK20]. However, such survey does not show the systematic usage of simulations to test ML models on synthetic problem. This work lists a few instances of the use of synthetic data, to detect conceptual errors in ML models.

> Our simulation-based testing can be seen as a special case in general computer science that focuses on empirical research in software engineering. We did not evaluate our method in a bigger context.

MSR/ESE When moving towards MSR/ESE, advice again becomes more similar to general statistic science, including many references to statistic science. A typical reference is a paper on events per variable (EPV) [PCK+96], original to medicine, which appears as reference in many ESE/MSR studies using logistic regression (e.g., in [GdCZ19, JTH21, TH18, TMHM17, PFD11]).

> In one of our simulation-based tests, we have shown how to tailor such findings towards MSR/ESE scenarios by own simulations.

Authors of [RDCJ18] report on the occurrence of well-known threats in existing literature.

> Opposed to a plain literature survey, our simulation-based testing is a high-order method used to clearly disambiguate threats to methods. It helps to better understand methods and threats.

In [BRB+09, KGB+14, KGB+16], authors focus on methodological advice when studying GitHub as a primary data source. Benefits and pitfalls, discussed in this paper, can be considered as guidelines for the data analysis targeting GitHub. In [KAB+08], the authors propose a reading method to evaluate guidelines for empirical software engineering. Authors report on their experience in defect modeling in [TH18]. Methodological challenges of highly correlated variables in defect models are discussed in [JTH21]. Methods to do model validation, including cross-validation variants, are examined in [TMHM17].

> All such paper have in common that they aim to give very general methodological advice for MSR/ESE. However, they limit their examination of methods to conceptual discussions, or to running them or real data. Such practice does not facilitate some conclusions on methods, including the potential impact of assumptions, alternative results, or threats. Simulation may help to understand how methods operate in a synthetic scenario. To the best of our knowledge, there is no (high-order) testing method in MSR/ESE like this, dedicated to the methods used in a study.

The distinction between simulation, inference, and prediction is often vague. This gets clear in work like [dFA20], that discuses the role of simulation-based studies in software engineering. The authors primarily discuss simulation-based studies as another method to produce empirical results.

> We consider a simulation as a high-order method to better understand, test and operationalize insights about methods used in MSR/ESE. We show detailed cases how simulations help to revise published methods.

In the following, we list work in MSR/ESE that refers to their own approach as simulation.

In [HHG14, HHH+15, Hon15], simulations of the software development process are introduced to help project managers to extrapolate future scenarios. Data mined from repositories is used to construct the simulations. The authors use agent-based systems. In such case, simulations are used to extrapolate, which is reasonable if configured with the right prior knowledge on unobserved variables. In [SL09], agent-based simulations for OS development are created using prior literature to set the relevant unobserved variables. In [BBH+19], multi-agent simulations predict next moves of agents. In [BSSG20], social coding dynamics are simulated based on historic data to forecast information spread.

> In contrast to such work, our simulations test methods used in MSE/ESE studies.

8.3 Empirical Contributions

This section focuses on concrete empirical work in MSR/ESE and how our technical and methodological contributions may guide their improvement. For a more detailed

overview on empirical studies on GitHub, we refer to [CIC16].

Recent studies contain the empirical analysis of API migration [SPN+18, RvDV17, RvDV12, RvDV13], developers experience [RRC16, RD11, ETL11], software changes [MPS08, SZZ05, YMNC04, KYM06, MSR17], entropy of changes [Has09], infrastructure as code [OZR22, OZVR21], dependencies [SPN+18, HV15, OBL10, RvDV13], network metrics [ZN08], diversity [VPR+15], similarity [LKMZ12, APM04, CCP07, MMWB13, SL16, HAL18], architecture [LLN14], documentation [AHS14], source code [ETL11, CDR18, FOMM10, GKSD05, CCP07], static code attributes [MGF07], change bursts [NZZ+10, Cho20], corrective engineering, bugs, defects and fixes [MW00, MGF07, RD11, Has09, ZN08, MPS08, SZZ05, NZZ+10, FBF+20], commit time [ETL11, SZZ05], pull requests [GPvD14, BPWS20, GZSvD15, GSB16, YWF+15, TDH14], open-source collaboration [Cho20], branching [KPB18], tests [BFS+18], OO-metrics [BBM96], asserts [CDO+15, KL17], social factors [FBF+20, VPR+15] model-driven technologies [KMK+15, HHL18, RHC+19, RHH+17], project popularity [AHS14, BHV16, WL14], languages usage [BTL+13, SHL+19], software builds [MAH10, GdCZ19] or reviewer assignment [RRC16, SdLJPM18].

There are some studies that already rely on dedicated solutions to scale analysis, like Boa (see [DRNN14]). However, several of them are also written in general purpose languages (e.g., see [SPN+18] which needs 4 weeks for analyzing revisions of 291 repositories).

> All such studies may benefit from scalable methods, like from our incremental map-reduce method, but also from DJ-Rex, Boa and LISA. We also assume that most of them may benefit from a better understanding of the underlying technology stacks, which is often heterogeneous in the practice.

From a methodological side, we are not aware of empirical studies in MSR/ESE that test a methodology by simulations and report on this. In the following, we discuss some studies that may potentially benefit from simulation-based testing.

In [GdCZ19], reasons for long duration builds in continuous integration pipelines are examined using multilevel models. Boh et al. [BSE07] shows an effect of experience on productivity using multilevel models. In [VPR+15], multilevel models are used to examine gender and tenure diversity. The authors of the previous papers are aware of the issues of dependent observations using advanced solutions, not comparable to the methodology shown in our first case (Sec. 5.4.2).

> However, multilevel models are complicated. Our experience is that simulations can be a great help in testing and understanding how multilevel models react to the threat of a structured sampling process in MSR/ESE.

Several works in MSR/ESE uses a methodology that assumes completely independent observations and thereby invokes threats (e.g., [RD11, TBP+17, ZHMZ17]).

> Such work may benefit from simulating the structured sampling process, and other reoccurring structural entities, like artifacts and developers, to recognize the potential dangers for results.

There is work discussing aggregation or disaggregation strategies applied to software engineering data [ZHMZ17, HZ13].

> In simulations, it is easy to show that aggregation artificially increases correlation. Simulation-based tests may guide novel ideas on how to resolve issues with correlated variables (Sec. **??**), potentially by disaggregated analysis of repository data.

Further, we assume that a series of work, relying on the well known SZZ algorithm [SZZ05], may benefit from simulation-based testing. Defect classification produced by SZZ is critically influenced by the sampling process, and the temporal evolution of commits in a repository.

> We assume that simulations of commit and fix behavior of developers can easily uncover that SZZ classifications share a natural correlation with time because for later commits, opportunities being fixed are just getting rare. This can be considered as a systematic measurement error. Hence, the effect of every metric correlating with time, e.g., experience measures, may be confused with such effect. It may be resolved in parts by the control of variables (Sec. **??**).

Bird at al. [BBA+09] examine the empirical challenges of incorrectly labeled bugs in historical defect data, which is an important threat to following up methodology.

> Transferring this reference to our terminology, a 'fix' is an observed variable, but the actual 'bug' is unobserved. We may simulate both to examine the impact of different assumption on this relation. Bird at al. does an initial step in the examination, but does not use synthetic fix-bug-pairs. This makes forming a precise picture complicated.

Chapter 9

Conclusion

Summary Empirical studies in software engineering use software repositories as data sources to understand software development. Repository data is either used to answer questions that guide the decision-making in the software development, or to provide tools that help with practical aspects of developers' everyday work. Studies are classified into the field of Empirical Software Engineering (ESE), and more specifically into Mining Software Repositories (MSR).

Studies working with repository data often focus on their results. Results are statements or tools, derived from the data, that help with practical aspects of software development. This thesis focuses on the methods and high-order methods used to produce such results. In particular, we focus on incremental methods to scale the processing of repositories, declarative methods to compose a heterogeneous analysis, and high-order methods used to reason about threats to methods operating on repositories. We summarize this as technical and methodological improvements. We contribute the improvements to methods and high-order methods in the context of MSR/ESE to produce future empirical results more effectively. We contribute the following improvements.

Chapter 3 We propose a method to improve the scalability of functions that abstract over repositories with high revision count in a theoretically founded way. We use insights on abstract algebra and program incrementalization to define a core interface of high-order functions that compute scalable static abstractions of a repository with many revisions. We evaluate the scalability of our method by benchmarks, comparing a prototype with available competitors in MSR/ESE.

This contribution found on a strong theoretical background and applies it to the study of repository history, which is a novel application field. The placeholder used to illustrate this new method is to compute the cyclomatic complexity metrics for a repository. This is a rather simplistic showcase. However, the proposed map-reduce interface of our method can be applied broadly. The main limitations of our method are the missing support for recursion, and the less-modular definition of an analysis, given in terms of chained map-reduce function applications. The popularity of map-reduce can be seen as an indication of its usability.

Chapter 4 We propose a method to improve the definition of functions that abstract over a repository with a heterogeneous technology stack, by using concepts

from declarative logic programming and combining them with ideas on megamodeling and linguistic architecture. We reproduce existing ideas on declarative logic programming with languages close to Datalog, coming from architecture recovery, source code querying, and static program analysis, and transfer them from the analysis of a homogeneous to a heterogeneous technology stack. We provide a prove-of-concept of such method in a case study.

The proposed method using Datalog differs from the map-reduce method, presented in Chapter 3. Datalog allows composing analysis of repositories in terms of modular rules. The modular rules require a schema on how to structure the data extracted by the rules. We borrow this schema from previous work on linguistic architecture and megamodeling. A prove-of-concept shows that such analysis of a heterogeneous technology stack is possible. The placeholder used to demonstrate our method is analyzing EMF patterns of usage. Compared to studying the cyclomatic complexity, studying EMF patterns of usage is technically more demanding. The main limitation of our Datalog method is the scalability, which we did not examine in detail. Scientific applications of declarative languages, close to Datalog, used for related purposes (architecture recovery, source code querying and static program analysis), can be seen as an indication of its usability.

Chapter 5 We propose a high-order method to improve the disambiguation of threats to methods used in MSR/ESE. We focus on a better disambiguation of threats, operationalizing reasoning about them, and making the implications to a valid data analysis methodology explicit, by using simulations. We encourage researchers to accomplish their work by implementing 'fake' simulations of their MSR/ESE scenarios, to operationalize relevant insights about alternative plausible results, negative results, potential threats and the used data analysis methodologies. We prove that such way of simulation-based testing contributes to the disambiguation of threats in published MSR/ESE research.

We consider the proposed method to use simulations to be a high-order method. We use it to better understand sets of methods (the methodology) in studies in MSR/ESE.

We assume that the technical and methodological improvements, presented in this thesis, will help future studies to produce new empirical results more effectively. We consider the following aspects to be the relevant future work on the challenges discussed in the thesis.

Future Work on Scalability Relying solely on adding hardware in a distributed setup should not be the only option to scale data processing, if there are other options, like incrementalization, or memoization. In the field of MSR/ESE, we have identified a less resource-intensive option to compute abstractions of repositories. This is presented in Chapter 3. Methods for the technical analysis should work with such advantage. We assume that the following mostly scalability related points are relevant for the future.

- In Chapter 4, we have used Datalog as an alternative to map-reduce. The realization, that we have presented so far, is a prove-of-concept. Datalog should be a subject to future work on improving the scalability too. Datalog

allows incremental execution. We are confident that ideas on incremental static analysis with languages close to Datalog can successfully be transferred to the processing of repository history.

- One question that we did not address is how efficient data storage and archiving methods can be used to back our methods. We limit our methods to the computation of scalable abstractions, but do not consider storage. We currently use default storage mechanisms and access to cloned GitHub repositories. We may benefit from access to GHTorrent or Software Heritage. Depending on which data we process, we may also benefit from graph compression.

- Scalable computation of abstractions, as we discuss it, mostly ends with a dataset. We typically call this methodological step the data collection. The dataset can then be used to build more elaborate abstractions. Often, such abstractions are more complex models of the software engineering process. We assume that such strict line between models and data collection is outdated (at least in the field of MSR). A tighter connection can be beneficial. Scalability may be improved by navigating the extraction of repository data by a model's understanding of importance.

Future Work on Heterogeneity As a prove-of-concept, we illustrate our declarative method in a case study on EMF usage patterns. We assume that the combination of modular rules with a well-defined schema for structuring knowledge on technologies and languages, is an alternative to the usage of chains of map-reduce calls to analyze repositories. However, evaluating the benefits of such declarative programming style, applied in a heterogeneous context, still needs to be done more systematically. We assume that the following mostly heterogeneity related points are relevant for the future.

- We need a more systematic way of comparing the benefits of Datalog and other (non-) declarative languages, used in a heterogeneous MSR/ESE setting. The appropriate way to do so is a randomized experiment. This is very difficult due to the complexity of typical MSR/ESE tasks. Finding subjects to perform such tasks and potentially training the subjects in advance of an experiment is challenging. We find the same problem in related work that proposes methods to mine repositories. We are not aware of any work proving the usability of methods in experiments. Experiments may be needed in the future.

- The usage of our declarative solution calls for a catalog of rules to abstract over different technologies. Such a catalog may be designed in close analogy to previous work on megamodeling and linguistic architecture. It may be relevant to decouple and reuse efforts on abstracting over existing technologies. It may also be an alternative to having fixed data sets, which are essentially materialized abstractions. Instead of a dataset, MSR/ESE work may share their modular rules in that also other authors can benefit from the mining logic, and can reproduce it on new or the same repositories. Reusing monolithic functionality is typically more complicated than reusing modular functionality.

- A prerequisite for our modular mining is a schema that we borrow from previous work on megamodeling and linguistic architecture. Understanding in how far the current schema is suited to abstract over complex technology, still requires future work.

Future Work on Validity The implications and possibilities of the simulation-based testing method are wide. In our presentation, we have started showing typical threats of methods that are used in existing studies in MSE/ESE. We assume that the following mostly validity related points are relevant for the future.

- We suggest forming a catalog of simulations that make threats to methods more explicit. This can be a step towards better understanding and testing the methods used in the scientific process. This may assure that future studies improve, and it can serve as a counteraction against reproducibility crisis.
- The catalog can be seen as a part of an MSR/ESE body of knowledge, that can serve as a reference for studies and for teaching. Simulations recently proved their importance for teaching in the recent statistic literature.
- We assume that studies in MSR/ESE need to be more careful in distinguishing between results, methods and high-order methods, when doing empirical research. If researchers contribute new results, the methods used to produce the results should be valid. If researchers contribute new methods, the results should be seen critically. Intuitive results should not be taken as the validation of a set of methods. Simulations may be a better option for validating a set of methods. Presenting a new method, with new results, can be a mistake. Researchers may modify the methods until the results fit their intuition.

Bibliography

[AA17] Mattia Atzeni and Maurizio Atzori. CodeOntology: RDF-ization of Source Code. In *Proc. ISWC*, 2017.

[AB11] Andrea Arcuri and Lionel C. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE*, pages 1–10. ACM, 2011.

[AC14] Özlem Albayrak and Jeffrey C. Carver. Investigation of individual factors impacting the effectiveness of requirements inspections: a replicated experiment. *Empir. Softw. Eng.*, 19(1):241–266, 2014.

[ACC+10] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *EuroSys*, pages 223–236. ACM, 2010.

[ADGC14] Cecilia Apa, Oscar Dieste, Edison G. Espinosa G., and Efraín R. Fonseca C. Effectiveness for detecting faults within and outside the scope of testing techniques: an independent replication. *Empir. Softw. Eng.*, 19(2):378–417, 2014.

[ADT+18] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *SIGMOD Conference*, pages 601–613. ACM, 2018.

[AEJO19] Mario Alvarez-Picallo, Alex Eyers-Taylor, Michael Peyton Jones, and C.-H. Luke Ong. Fixing Incremental Computation - Derivatives of Fixpoints, and the Recursive Semantics of Datalog. In *ESOP*, volume 11423 of *Lecture Notes in Computer Science*, pages 525–552. Springer, 2019.

[AG15] Carol V. Alexandru and Harald C. Gall. Rapid Multi-Purpose, Multi-Commit Code Analysis. In *ICSE (2)*, pages 635–638. IEEE Computer Society, 2015.

[Agg15] Charu C Aggarwal. *Data mining*, volume 1. Springer, 2015.

[Agh90] Gul A. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press, 1990.

[AHS14] Karan Aggarwal, Abram Hindle, and Eleni Stroulia. Co-evolution of project documentation and within github. In *MSR*, pages 360–363. ACM, 2014.

[Aka98] Hirotogu Akaike. *Information Theory and an Extension of the Maximum Likelihood Principle*, pages 199–213. Springer, 1998.

[AKM08] Abdulkareem Alali, Huzefa H. Kagdi, and Jonathan I. Maletic. What's a Typical Commit? A Characterization of Open Source Software Repositories. In *ICPC*, pages 182–191. IEEE Computer Society, 2008.

[AMLH14] Carlo Angiuli, Edward Morehouse, Daniel R. Licata, and Robert Harper. Homotopical patch theory. In *ICFP*, pages 243–256. ACM, 2014.

[APG17] Carol V. Alexandru, Sebastiano Panichella, and Harald C. Gall. Reducing redundancies in multi-revision code analysis. In *SANER*, pages 148–159. IEEE Computer Society, 2017.

[APM04] Giuliano Antoniol, Massimiliano Di Penta, and Ettore Merlo. An Automatic Approach to identify Class Evolution Discontinuities. In *IWPSE*, pages 31–40. IEEE Computer Society, 2004.

[APPG19] Carol V. Alexandru, Sebastiano Panichella, Sebastian Proksch, and Harald C. Gall. Redundancy-free analysis of multi-revision software artifacts. *Empirical Software Engineering*, 24(1):332–380, 2019.

[AS05] Bente Anda and Dag I. K. Sjøberg. Investigating the Role of Use Cases in the Construction of Class Diagrams. *Empir. Softw. Eng.*, 10(3):285–309, 2005.

[AXL+15] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational Data Processing in Spark. In *SIGMOD Conference*, pages 1383–1394. ACM, 2015.

[BAB+21] Bret Beheim, Quentin D. Atkinson, Joseph Bulbulia, Will Gervais, Russell D. Gray, Joseph Henrich, Martin Lang, M. Willis Monroe, Michael Muthukrishna, Ara Norenzayan, Benjamin Grant Purzycki, Azim Shariff, Edward Slingerland, Rachel Spicer, and Aiyana K Willard. Treatment of missing data determined conclusions regarding moralizing gods. *Nature*, 595(7866):1476–4687, 2021.

[BARH06] Andrea Burton, Douglas G Altman, Patrick Royston, and Roger L Holder. The design of simulation studies in medical statistics. *Statistics in Medicine*, 25(24):4279–4292, 2006.

[BBA+09] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar T. Devanbu. Fair and

balanced?: bias in bug-fix datasets. In *ESEC/SIGSOFT FSE*, pages 121–130. ACM, 2009.

[BBH+19] Jim Blythe, John Bollenbacher, Di Huang, Pik-Mai Hui, Rachel Krohn, Diogo Pacheco, Goran Muric, Anna Sapienza, Alexey Tregubov, Yong-Yeol Ahn, Alessandro Flammini, Kristina Lerman, Filippo Menczer, Tim Weninger, and Emilio Ferrara. Massive Multi-agent Data-Driven Simulations of the GitHub Ecosystem. In *PAAMS*, volume 11523 of *Lecture Notes in Computer Science*, pages 3–15. Springer, 2019.

[BBM96] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Trans. Software Eng.*, 22(10):751–761, 1996.

[BDB08] R Harald Baayen, Douglas J Davidson, and Douglas M Bates. Mixed-effects modeling with crossed random effects for subjects and items. *Journal of Memory and Language*, 59(4):390–412, 2008.

[Béz05a] Jean Bézivin. Model driven engineering: An emerging technical space. In *GTTSE*, volume 4143 of *Lecture Notes in Computer Science*, pages 36–64. Springer, 2005.

[Béz05b] Jean Bézivin. On the unification power of models. *Softw. Syst. Model.*, 4(2):171–188, 2005.

[BF05] Rajendra Bose and James Frew. Lineage retrieval for scientific data processing: a survey. *ACM Comput. Surv.*, 37(1):1–28, 2005.

[BFS+18] Neil C. Borle, Meysam Feghhi, Eleni Stroulia, Russell Greiner, and Abram Hindle. Analyzing the effects of test driven development in GitHub. *Empirical Software Engineering*, 23(4):1931–1958, 2018.

[BHV16] Hudson Borges, André C. Hora, and Marco Tulio Valente. Predicting the Popularity of GitHub Repositories. In *PROMISE*, pages 9:1–9:10. ACM, 2016.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Information science and statistics. Springer, 1st edition, 2006.

[BJRV05] Jean Bézivin, Frédéric Jouault, Peter Rosenthal, and Patrick Valduriez. Modeling in the Large and Modeling in the Small. In *Proc. MDAFA 2003 and MDAFA 2004*, volume 3599 of *LNCS*, pages 33–46. Springer, 2005.

[BJV04] Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. On the Need for Megamodels. In *Proc. OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop*, 2004.

[BK20] Houssem Ben Braiek and Foutse Khomh. On testing machine learning programs. *J. Syst. Softw.*, 164:110542, 2020.

[Bla77] Peter Michael Blau. *Inequality and heterogeneity: A primitive theory of social structure*, volume 7. Free Press New York, 1977.

[BLST13] Dale J. Barr, Roger Levy, Christoph Scheepers, and Harry J. Tily. Random effects structure for confirmatory hypothesis testing: Keep it maximal. *Journal of Memory and Language*, 368(3):255–278, 2013.

[BPVZ20] Paolo Boldi, Antoine Pietri, Sebastiano Vigna, and Stefano Zacchiroli. Ultra-large-scale repository analysis via graph compression. In *SANER*, pages 184–194. IEEE, 2020.

[BPWS20] Marcus Bertoncello, Gustavo Pinto, Igor Scaliante Wiese, and Igor Steinmacher. Pull Requests or Commits? Which Method Should We Use to Study Contributors' Behavior? In *SANER*. IEEE Computer Society, 2020.

[BRB+09] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. Germán, and Premkumar T. Devanbu. The promises and perils of mining git. In *MSR*, pages 1–10. IEEE Computer Society, 2009.

[BROL14] P. Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy J. Lin. Summingbird: A Framework for Integrating Batch and Online MapReduce Computations. *PVLDB*, 7(13):1441–1451, 2014.

[BS09] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, pages 243–262. ACM, 2009.

[BSE07] Wai Fong Boh, Sandra Slaughter, and J. Alberto Espinosa. Learning from Experience in Software Development: A Multilevel Analysis. *Manag. Sci.*, 53(8):1315–1331, 2007.

[BSHA20] Abdul Ali Bangash, Hareem Sahar, Abram Hindle, and Karim Ali. On the time-based conclusion stability of cross-project defect prediction models. *Empirical Software Engineering*, pages 1–38, 2020.

[BSSG20] Neda Hajiakhoond Bidoki, Madeline Schiappa, Gita Sukthankar, and Ivan Garibay. Modeling social coding dynamics with sampled historical data. *Online Soc. Networks Media*, 16:100070, 2020.

[BTL+13] Tegawendé F. Bissyandé, Ferdian Thung, David Lo, Lingxiao Jiang, and Laurent Réveillère. Popularity, Interoperability, and Impact of Programming Languages in 100, 000 Open Source Projects. In *COMPSAC*, pages 303–312. IEEE Computer Society, 2013.

[BWGW23] Marvin Muñoz Barón, Marvin Wyrich, Daniel Graziotin, and Stefan Wagner. Evidence profiles for validity threats in program comprehension experiments. In *ICSE*, pages 1907–1919. IEEE, 2023.

[BWR+11] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquini. Incoop: MapReduce for incremental computations. In *SoCC*, page 7. ACM, 2011.

[CBB+03] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Ying Xing, and Stanley B. Zdonik. Scalable Distributed Stream Processing. In *CIDR*. www.cidrdb.org, 2003.

[CCP07] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. Identifying Changed Source Code Lines from Version Repositories. In *MSR*, page 14. IEEE Computer Society, 2007.

[CCT09] James Cheney, Laura Chiticariu, and Wang Chiew Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[CCWA13] Jacob Cohen, Patricia Cohen, Stephen G West, and Leona S Aiken. *Applied multiple regression/correlation analysis for the behavioral sciences*. Routledge, 2013.

[CDO+15] Casey Casalnuovo, Premkumar T. Devanbu, Abílio Oliveira, Vladimir Filkov, and Baishakhi Ray. Assert Use in GitHub Projects. In *ICSE (1)*, pages 755–766. IEEE Computer Society, 2015.

[CDR18] Maximilian Capraro, Michael Dorner, and Dirk Riehle. The patchflow method for measuring inner source collaboration. In *MSR*, pages 515–525. ACM, 2018.

[CGRO14] Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. A theory of changes for higher-order languages: incrementalizing λ-calculi by static differentiation. In *PLDI*, pages 145–155. ACM, 2014.

[Cho20] *Using Productive Collaboration Bursts to Analyze Open Source Collaboration Effectiveness*. IEEE, 2020.

[CIC16] Valerio Cosentino, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Findings from GitHub: methods, datasets and limitations. In *Proc. MSR*, pages 137–141, 2016.

[CJ18] Haipeng Cai and John Jenkins. Leveraging historical versions of Android apps for efficient and precise taint analysis. In *MSR*, pages 265–269. ACM, 2018.

[CLP+15] Gerardo Canfora, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Defect prediction as a multiobjective optimization problem. *Softw. Test. Verification Reliab.*, 25(4):426–459, 2015.

[CSFG19] April Clyburne-Sherin, Xu Fei, and Seth Ariel Green. Computational reproducibility via containers in psychology. *Meta-psychology*, 3, 2019.

[CSS13] K. K. Chaturvedi, V. B. Singh, and Prashast Singh. Tools in Mining Software Repositories. In *ICCSA (6)*, pages 89–98. IEEE Computer Society, 2013.

[CY12] Rada Chirkova and Jun Yang. Materialized Views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.

[CZ17] Roberto Di Cosmo and Stefano Zacchiroli. Software Heritage: Why and How to Preserve Software Source Code. In *iPRES*, 2017.

[DAB21] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling Projects in GitHub for MSR Studies. In *MSR*, pages 560–564. IEEE, 2021.

[DB18] Annette J Dobson and Adrian G Barnett. *An introduction to generalized linear models*. CRC press, 2018.

[DBG+15] Martin Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. Untangling fine-grained code changes. In *SANER*, pages 341–350. IEEE Computer Society, 2015.

[DEGV01] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

[dFA20] Breno Bernard Nicolau de França and Nauman Bin Ali. The Role of Simulation-Based Studies in Software Engineering Research. In *Contemporary Empirical Methods in Software Engineering*, pages 263–287. Springer, 2020.

[DLR10] Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *MSR*, pages 31–41. IEEE Computer Society, 2010.

[DNRN13] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In *ICSE*, pages 422–431. IEEE Computer Society, 2013.

[DNRN15] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: Ultra-Large-Scale Software Repository and Source-Code Mining. *ACM Trans. Softw. Eng. Methodol.*, 25(1):7:1–7:34, 2015.

[dPSER20] Jens Van der Plas, Quentin Stiévenart, Noah Van Es, and Coen De Roover. Incremental Flow Analysis through Computational Dependency Reification. In *SCAM*, pages 25–36. IEEE, 2020.

[DRNN14] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. Mining billions of AST nodes to study actual and potential usage of Java language features. In *ICSE*, pages 779–790. ACM, 2014.

[EKKM08] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and continuous checking of structural program dependencies. In *ICSE*, pages 391–400. ACM, 2008.

[ELZ+10] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey C. Fox. Twister: a runtime for iterative MapReduce. In *HPDC*, pages 810–818. ACM, 2010.

[ETL11] Jon Eyolfson, Lin Tan, and Patrick Lam. Do time of day and developer experience affect commit bugginess. In *MSR*, pages 153–162. ACM, 2011.

[FBF+20] Filipe Falcão, Caio Barbosa, Baldoino Fonseca, Alessandro Garcia, Márcio Ribeiro, and Rohit Gheyi. On Relating Technical, Social Factors, and the Introduction of Bugs. In *SANER*, pages 378–388. IEEE, 2020.

[Feg16] Leonidas Fegaras. Incremental Query Processing on Big Data Streams. *IEEE Trans. Knowl. Data Eng.*, 28(11):2998–3012, 2016.

[FKMP05] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[FLHV22] Hongbo Fang, Hemank Lamba, James D. Herbsleb, and Bogdan Vasilescu. "this is damn slick!" estimating the impact of tweets on open source project popularity and new contributors. In *ICSE*, pages 2116–2129. ACM, 2022.

[FLV12] Jean-Marie Favre, Ralf Lämmel, and Andrei Varanovich. Modeling the Linguistic Architecture of Software Products. In *Proc. MODELS 2012*, volume 7590 of *LNCS*, pages 151–167. Springer, 2012.

[FOMM10] Thomas Fritz, Jingwen Ou, Gail C. Murphy, and Emerson R. Murphy-Hill. A degree-of-knowledge model to capture source code familiarity. In *ICSE (1)*, pages 385–394. ACM, 2010.

[For82] Charles Forgy. Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem. *Artif. Intell.*, 19(1):17–37, 1982.

[GAB+20] Alessandro Gasparini, Keith R Abrams, Jessica K Barrett, Rupert W Major, Michael J Sweeting, Nigel J Brunskill, and Michael J Crowther. Mixed-effects models for health care longitudinal data with an informative visiting process: A Monte Carlo simulation study. *Statistica Neerlandica*, 74(1):5–23, 2020.

[GCB+21] Felix Grund, Shaiful Alam Chowdhury, Nick C. Bradley, Braxton Hall, and Reid Holmes. CodeShovel: Constructing Method-Level Source Code Histories. In *ICSE*, pages 1510–1522. IEEE, 2021.

[GdCZ19] Taher Ahmed Ghaleb, Daniel Alencar da Costa, and Ying Zou. An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering*, 24(4):2102–2139, 2019.

[GGMS97] Dieter Gluche, Torsten Grust, Christof Mainberger, and Marc H. Scholl. Incremental Updates for Materialized OQL Views. In *DOOD*, volume 1341 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 1997.

[GH06] Andrew Gelman and Jennifer Hill. *Data analysis using regression and multilevel/hierarchical models*. Cambridge university press, 2006.

[GHJ98] Harald C. Gall, Karin Hajek, and Mehdi Jazayeri. Detection of Logical Coupling Based on Product Release History. In *ICSM*, pages 190–197. IEEE Computer Society, 1998.

[GHK+19] Bernardo Cuenca Grau, Ian Horrocks, Mark Kaminski, Egor V. Kostylev, and Boris Motik. Limit Datalog: A Declarative Query Language for Data Analysis. *SIGMOD Rec.*, 48(4):6–17, 2019.

[GHLZ13] Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. Datalog and Recursive Query Processing. *Found. Trends Databases*, 5(2):105–195, 2013.

[GHV20] Andrew Gelman, Jennifer Hill, and Aki Vehtari. *Regression and other stories*. Cambridge University Press, 2020.

[GKIT07] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update Exchange with Mappings and Provenance. In *VLDB*, pages 675–686. ACM, 2007.

[GKMS00] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey P. Siy. Predicting Fault Incidence Using Software Change History. *IEEE Trans. Software Eng.*, 26(7):653–661, 2000.

[GKSD05] Tudor Gîrba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How Developers Drive Software Evolution. In *IWPSE*, pages 113–122. IEEE Computer Society, 2005.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *SIGMOD Conference*, pages 157–166. ACM Press, 1993.

[Gou13] Georgios Gousios. The GHTorent dataset and tool suite. In *MSR*, pages 233–236. IEEE Computer Society, 2013.

[GPvD14] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *ICSE*, pages 345–355. ACM, 2014.

[GS10] Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *SIGSOFT FSE*, pages 147–156. ACM, 2010.

[GS12] Georgios Gousios and Diomidis Spinellis. GHTorrent: Github's data from a firehose. In *MSR*, pages 12–21. IEEE Computer Society, 2012.

[GSB16] Georgios Gousios, Margaret-Anne D. Storey, and Alberto Bacchelli. Work practices and challenges in pull-based development: the contributor's perspective. In *ICSE*, pages 285–296. ACM, 2016.

[GST90] Sumit Ganguly, Abraham Silberschatz, and Shalom Tsur. A Framework for the Parallel Processing of Datalog Queries. In *SIGMOD Conference*, pages 143–152. ACM Press, 1990.

[GSV16] Georgios Gousios, Dominik Safaric, and Joost Visser. Streaming software analytics. In *BIGDSE@ICSE*, pages 8–11. ACM, 2016.

[GZSvD15] Georgios Gousios, Andy Zaidman, Margaret-Anne D. Storey, and Arie van Deursen. Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective. In *ICSE*, pages 358–368. IEEE Computer Society, 2015.

[HAL18] Johannes Härtel, Hakan Aksu, and Ralf Lämmel. Classification of APIs by hierarchical clustering. In *ICPC*, pages 233–243. ACM, 2018.

[Har15] Frank E Harrell. *Regression modeling strategies: with applications to linear models, logistic and ordinal regression, and survival analysis*, volume 2. Springer, 2015.

[Has06] Ahmed E. Hassan. Mining Software Repositories to Assist Developers and Support Managers. In *ICSM*, pages 339–342. IEEE Computer Society, 2006.

[Has08] Ahmed E. Hassan. The road ahead for Mining Software Repositories. In *Frontiers of Software Maintenance*, pages 48–57, 2008.

[Has09] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *ICSE*, pages 78–88. IEEE, 2009.

[Hei22] Marcel Heinz. Knowledge engineering for software languages and software technologies. 2022.

[HGL11] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: an interactive tutorial. In *SIGMOD Conference*, pages 1213–1216. ACM, 2011.

[HHG14] Verena Honsel, Daniel Honsel, and Jens Grabowski. Software Process Simulation Based on Mining Software Repositories. In *ICDM Workshops*, pages 828–831. IEEE Computer Society, 2014.

[HHH+15] Verena Honsel, Daniel Honsel, Steffen Herbold, Jens Grabowski, and Stephan Waack. Mining Software Dependency Networks for Agent-Based Simulation of Software Evolution. In *ASE Workshops*, pages 102–108. IEEE Computer Society, 2015.

[HHK20] Yoshiki Higo, Shinpei Hayashi, and Shinji Kusumoto. On tracking Java methods with Git mechanisms. *J. Syst. Softw.*, 165:110571, 2020.

[HHL+17] Johannes Härtel, Lukas Härtel, Ralf Lämmel, Andrei Varanovich, and Marcel Heinz. Interconnected Linguistic Architecture. *Programming Journal*, 1(1):3, 2017.

[HHL18] Johannes Härtel, Marcel Heinz, and Ralf Lämmel. EMF Patterns of Usage on GitHub. In *ECMFA*, volume 10890 of *Lecture Notes in Computer Science*, pages 216–234. Springer, 2018.

[HHL20] Marcel Heinz, Johannes Härtel, and Ralf Lämmel. Reproducible Construction of Interconnected Technology Models for EMF Code Generation. *J. Object Technol.*, 19(2):8:1–25, 2020.

[HK06] Imed Hammouda and Kai Koskimies. Concern based mining of heterogeneous software repositories. In *MSR*, pages 80–86. ACM, 2006.

[HL20] Johannes Härtel and Ralf Lämmel. Incremental Map-Reduce on Repository History. In *SANER*, pages 320–331. IEEE, 2020.

[HL22] Johannes Härtel and Ralf Lämmel. Operationalizing threats to MSR studies by simulation-based testing. In *MSR*, pages 86–97. IEEE, 2022.

[HL23] Johannes Härtel and Ralf Lämmel. Operationalizing validity of empirical software engineering studies. *Empir. Softw. Eng.*, 28(6):153, 2023.

[HLV17] Marcel Heinz, Ralf Lämmel, and Andrei Varanovich. Axioms of linguistic architecture. In *Proc. MODELSWARD 2017*, 2017.

[Hon15] Verena Honsel. Statistical Learning and Software Mining for Agent Based Simulation of Software Evolution. In *ICSE (2)*, pages 863–866. IEEE Computer Society, 2015.

[HPMY13] Zhimin He, Fayola Peters, Tim Menzies, and Ye Yang. Learning from Open-Source Projects: An Empirical Study on Defect Prediction. In *ESEM*, pages 45–54. IEEE Computer Society, 2013.

[HS13] Matthew Hayes and Sam Shah. Hourglass: A library for incremental processing on Hadoop. In *BigData*, pages 742–752. IEEE Computer Society, 2013.

[HV15] André C. Hora and Marco Tulio Valente. apiwave: Keeping track of API popularity and migration. In *ICSME*, pages 321–323. IEEE Computer Society, 2015.

[HVdM06] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. *codeQuest:* Scalable Source Code Queries with Datalog. In *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2006.

[HZ13] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *MSR*, pages 121–130. IEEE Computer Society, 2013.

[IR15] Guido W Imbens and Donald B Rubin. *Causal inference in statistics, social, and biomedical sciences*. Cambridge University Press, 2015.

[IYNH19] Rahul N Iyer, S Alex Yun, Meiyappan Nagappan, and Jesse Hoey. Effects of Personality Traits on Pull Request Acceptance. *IEEE Transactions on Software Engineering*, 2019.

[JLY+19] John Johnson, Sergio Lubo, Nishitha Yedla, Jairo Aponte, and Bonita Sharif. An Empirical Study Assessing Source Code Readability in Comprehension. In *ICSME*, pages 513–523. IEEE, 2019.

[JM15] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[JMF14] Ahmad Jbara, Adam Matan, and Dror G. Feitelson. High-MCC Functions in the Linux Kernel. *Empir. Softw. Eng.*, 19(5):1261–1298, 2014.

[JSD+20] Rodi Jolak, Maxime Savary-Leblanc, Manuela Dalibor, Andreas Wortmann, Regina Hebig, Juraj Vincur, Ivan Polásek, Xavier Le Pallec, Sébastien Gérard, and Michel R. V. Chaudron. Software engineering whispers: The effect of textual vs. graphical software design descriptions on software design communication. *Empir. Softw. Eng.*, 25(6):4427–4471, 2020.

[JTH21] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and Ahmed E. Hassan. The Impact of Correlated Metrics on the Interpretation of Defect Models. *IEEE Trans. Software Eng.*, 47(2):320–331, 2021.

[KAB+08] Barbara A. Kitchenham, Hiyam Al-Kilidar, Muhammad Ali Babar, Mike Berry, Karl Cox, Jacky Keung, Felicia Kurniawati, Mark Staples, He Zhang, and Liming Zhu. Evaluating guidelines for reporting empirical software engineering studies. *Empir. Softw. Eng.*, 13(1):97–121, 2008.

[KG11] Siim Karus and Harald C. Gall. A study of language usage evolution in open source software. In *MSR*, pages 13–22. ACM, 2011.

[KGB+14] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. Germán, and Daniela E. Damian. The promises and perils of mining GitHub. In *MSR*, pages 92–101. ACM, 2014.

[KGB+16] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. Germán, and Daniela E. Damian. An in-depth study of the promises and perils of mining GitHub. *Empir. Softw. Eng.*, 21(5):2035–2071, 2016.

[KL17] Pavneet Singh Kochhar and David Lo. Revisiting Assert Use in GitHub Projects. In *EASE*, pages 298–307. ACM, 2017.

[KMK+15] Dimitrios S. Kolovos, Nicholas Drivalos Matragkas, Ioannis Korkontzelos, Sophia Ananiadou, and Richard F. Paige. Assessing the Use of Eclipse MDE Technologies in Open-Source Software Projects. In *OSS4MDE@MoDELS*, volume 1541 of *CEUR Workshop Proceedings*, pages 20–29. CEUR-WS.org, 2015.

[KPB18] Vladimir Kovalenko, Fabio Palomba, and Alberto Bacchelli. Mining file histories: should we consider branches? In *ASE*, pages 202–213. ACM, 2018.

[KSA+13] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A Large-Scale Empirical Study of Just-in-Time Quality Assurance. *IEEE Trans. Software Eng.*, 39(6):757–773, 2013.

[KSW+13] Angelika Kusel, Johannes Schoenboeck, Manuel Wimmer, Werner Retschitzegger, Wieland Schwinger, and Gerti Kappel. Reality Check for Model Transformation Reuse: The ATL Transformation Zoo Case Study. In *Proc. AMT 2013*, volume 1077 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[KYM06] Huzefa H. Kagdi, Shehnaaz Yusuf, and Jonathan I. Maletic. Mining sequences of changed-files from version histories. In *MSR*, pages 47–53. ACM, 2006.

[Läm] Ralf Lämmel. *Software Languages Syntax, Semantics, and Metaprogramming*. Springer.

[LDKBJ22] Quentin Le Dilavrec, Djamel Eddine Khelladi, Arnaud Blouin, and Jean-Marc Jézéquel. HyperAST: Enabling Efficient Analysis of Software Histories at Scale. In *ASE*. IEEE Computer Society, 2022. To appear.

[Len02] Maurizio Lenzerini. Data Integration: A Theoretical Perspective. In *PODS*, pages 233–246. ACM, 2002.

[LKMZ12] Thierry Lavoie, Foutse Khomh, Ettore Merlo, and Ying Zou. Inferring Repository File Structure Modifications Using Nearest-Neighbor Clone Detection. In *WCRE*, pages 325–334. IEEE Computer Society, 2012.

[LLC+11] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. Parallel data processing with MapReduce: a survey. *SIGMOD Record*, 40(4):11–20, 2011.

[LLN14] Mircea Lungu, Michele Lanza, and Oscar Nierstrasz. Evolutionary and collaborative software architecture recovery with Softwarenaut. *Sci. Comput. Program.*, 79:204–223, 2014.

[LOR+10] Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin C. Webb, and Ken Yocum. Stateful bulk processing for incremental analytics. In *SoCC*, pages 51–62. ACM, 2010.

[LPS11] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. Large-scale, AST-based API-usage analysis of open-source Java projects. In *SAC*, pages 1317–1324. ACM, 2011.

[LSBV17] Davy Landman, Alexander Serebrenik, Eric Bouwers, and Jurgen J. Vinju. Corrigendum: Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions published on 9 December 2015. *J. Softw. Evol. Process.*, 29(10), 2017.

[LV14] Ralf Lämmel and Andrei Varanovich. Interpretation of Linguistic Architecture. In *Proc. ECMFA 2014*, volume 8569 of *LNCS*, pages 67–82. Springer, 2014.

[LW97] Leonid Libkin and Limsoon Wong. Query Languages for Bags and Aggregate Functions. *J. Comput. Syst. Sci.*, 55(2):241–272, 1997.

[LWL+05] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *PODS*, pages 1–12. ACM, 2005.

[MAB+10] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146. ACM, 2010.

[MAH10] Shane McIntosh, Bram Adams, and Ahmed E. Hassan. The evolution of ANT build systems. In *MSR*, pages 42–51. IEEE Computer Society, 2010.

[MB20] Ian R. McChesney and Raymond R. Bond. Observations on the Linear Order of Program Code Reading Patterns in Programmers with Dyslexia. In *EASE*, pages 81–89. ACM, 2020.

[McE20] Richard McElreath. *Statistical rethinking: A Bayesian course with examples in R and Stan.* CRC press, 2020.

[MG13] Samuel Mimram and Cinzia Di Giusto. A Categorical Theory of Patches. *Electr. Notes Theor. Comput. Sci.*, 298:283–307, 2013.

[MGF07] Tim Menzies, Jeremy Greenwald, and Art Frank. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Trans. Software Eng.*, 33(1):2–13, 2007.

[MHJ+15] William J. Martin, Mark Harman, Yue Jia, Federica Sarro, and Yuanyuan Zhang. The App Sampling Problem for App Store Mining. In *MSR*, pages 123–133. IEEE Computer Society, 2015.

[Mil06] Greg Miller. A Scientist's Nightmare: Software Problem Leads to Five Retractions. *Science*, 314(5807):1856–1857, 2006.

[MKPR11] Anne Martens, Heiko Koziolek, Lutz Prechelt, and Ralf H. Reussner. From monolithic to component-based performance evaluation of software architectures - A series of experiments analysing accuracy and effort. *Empir. Softw. Eng.*, 16(5):587–622, 2011.

[MMW02] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. *Expert Syst. Appl.*, 23(4):405–413, 2002.

[MMWB13] Xiaozhu Meng, Barton P. Miller, William R. Williams, and Andrew R. Bernat. Mining Software Repositories for Accurate Authorship. In *ICSM*, pages 250–259. IEEE Computer Society, 2013.

[Moc10] Audris Mockus. Organizational volatility and its effects on software defects. In *SIGSOFT FSE*, pages 117–126. ACM, 2010.

[MPS08] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *ICSE*, pages 181–190. ACM, 2008.

[MSR17] Tim Molderez, Reinout Stevens, and Coen De Roover. Mining change histories for unknown systematic edits. In *MSR*, pages 248–256. IEEE Computer Society, 2017.

[MT01] Tom Mens and Tom Tourwé. A Declarative Evolution Framework for Object-Oriented Design Patterns. In *ICSM*, pages 570–579. IEEE Computer Society, 2001.

[MW00] Audris Mockus and David M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[MWC19] Tim P Morris, Ian R White, and Michael J Crowther. Using simulation studies to evaluate statistical methods. *Statistics in Medicine*, 38(11):2074–2102, 2019.

[NBKO21] Sebastian Nielebock, Paul Blockhaus, Jacob Krüger, and Frank Ortmeier. AndroidCompass: A Dataset of Android Compatibility Checks in Code Repositories. In *MSR*, pages 535–539. IEEE, 2021.

[NDNR14] Hoan Anh Nguyen, Robert Dyer, Tien N. Nguyen, and Hridesh Rajan. Mining preconditions of APIs in large-scale code corpus. In *SIGSOFT FSE*, pages 166–177. ACM, 2014.

[NFK+18] Jaechang Nam, Wei Fu, Sunghun Kim, Tim Menzies, and Lin Tan. Heterogeneous Defect Prediction. *IEEE Trans. Software Eng.*, 44(9):874–896, 2018.

[NPK13] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. Transfer defect learning. In *ICSE*, pages 382–391. IEEE Computer Society, 2013.

[NZZ+10] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. Change Bursts as Defect Predictors. In *ISSRE*, pages 309–318. IEEE Computer Society, 2010.

[OBL10] Joel Ossher, Sushil Krishna Bajracharya, and Cristina Videira Lopes. Automated dependency resolution for open source software. In *MSR*, pages 130–140. IEEE Computer Society, 2010.

[OZR22] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. Smelly Variables in Ansible Infrastructure Code: Detection, Prevalence, and Lifetime. In *MSR*, pages 61–72. IEEE, 2022.

[OZVR21] Ruben Opdebeeck, Ahmed Zerouali, Camilo Velázquez-Rodríguez, and Coen De Roover. On the practice of semantic versioning for Ansible galaxy roles: An empirical study and a change classification model. *J. Syst. Softw.*, 182:111059, 2021.

[PBYI09] Lucian Popa, Mihai Budiu, Yuan Yu, and Michael Isard. DryadInc: Reusing Work in Large-scale Computations. In *HotCloud*. USENIX Association, 2009.

[PCGA08] Massimiliano Di Penta, Luigi Cerulo, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An empirical study of the relationships between design pattern roles and class change proneness. In *ICSM*, pages 217–226. IEEE Computer Society, 2008.

[PCK+96] Peter Peduzzi, John Concato, Elizabeth Kemper, Theodore R Holford, and Alvan R Feinstein. A simulation study of the number of events per variable in logistic regression analysis. *Journal of clinical epidemiology*, 49(12):1373–1379, 1996.

[PFD11] Daryl Posnett, Vladimir Filkov, and Premkumar T. Devanbu. Ecological inference in empirical software engineering. In *ASE*, pages 362–371. IEEE Computer Society, 2011.

[PFS+20] Valentina Piantadosi, Fabiana Fierro, Simone Scalabrino, Alexander Serebrenik, and Rocco Oliveto. How does code readability change during software evolution? *Empir. Softw. Eng.*, 25(6):5374–5412, 2020.

[PML15] Luca Ponzanelli, Andrea Mocci, and Michele Lanza. Summarizing Complex Development Artifacts by Mining Heterogeneous Data. In *MSR*, pages 401–405. IEEE Computer Society, 2015.

[PZS+20] Jevgenija Pantiuchina, Fiorella Zampetti, Simone Scalabrino, Valentina Piantadosi, Rocco Oliveto, Gabriele Bavota, and Massimiliano Di Penta. Why Developers Refactor Source Code: A Mining-based Study. *ACM Trans. Softw. Eng. Methodol.*, 29(4):29:1–29:30, 2020.

[QGMW96] Dallan Quass, Ashish Gupta, Inderpal Singh Mumick, and Jennifer Widom. Making Views Self-Maintainable for Data Warehousing. In *PDIS*, pages 158–169. IEEE Computer Society, 1996.

[RBC+17] David R. Roberts, Volker Bahn, Simone Ciuti, Mark S. Boyce, Jane Elith, Gurutzeta Guillera-Arroita, Severin Hauenstein, José J. Lahoz-Monfort, Boris Schröder, Wilfried Thuiller, David I. Warton, Brendan A. Wintle, Florian Hartig, and Carsten F. Dormann. Cross-validation strategies for data with temporal, spatial, hierarchical, or phylogenetic structure. *Ecography*, 40(8):913–929, 2017.

[RD11] Foyzur Rahman and Premkumar T. Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *ICSE*, pages 491–500. ACM, 2011.

[RDCJ18] Rolando P. Reyes, Oscar Dieste, Efraín R. Fonseca C., and Natalia Juristo. Statistical errors in software engineering experiments: a preliminary literature review. In *ICSE*, pages 1195–1206. ACM, 2018.

[RHC+19] Adithya Raghuraman, Truong Ho-Quang, Michel R. V. Chaudron, Alexander Serebrenik, and Bogdan Vasilescu. Does UML modeling associate with lower defect proneness?: a preliminary empirical investigation. In *MSR*, pages 101–104. IEEE / ACM, 2019.

[RHH+17] Gregorio Robles, Truong Ho-Quang, Regina Hebig, Michel R. V. Chaudron, and Miguel Angel Fernández. An extensive dataset of UML models in GitHub. In *Proc. MSR*, pages 519–522, 2017.

[RLP13] Coen De Roover, Ralf Lämmel, and Ekaterina Pek. Multi-dimensional exploration of API usage. In *Proc. ICPC 2013*, pages 152–161. IEEE, 2013.

[RNKJ11] Coen De Roover, Carlos Noguera, Andy Kellens, and Viviane Jonckers. The SOUL tool suite for querying programs in symbiosis with Eclipse. In *PPPJ*, pages 71–80. ACM, 2011.

[Roo11] Coen De Roover. A logic meta-programming foundation for example-driven pattern detection in object-oriented programs. In *Proc. ICSM*, pages 556–561. IEEE, 2011.

[RPD12] Foyzur Rahman, Daryl Posnett, and Premkumar T. Devanbu. Recalling the "imprecision" of cross-project defect prediction. In *SIGSOFT FSE*, page 61. ACM, 2012.

[RR93] G. Ramalingam and Thomas W. Reps. A Categorized Bibliography on Incremental Computation. In *POPL*, pages 502–510. ACM Press, 1993.

[RRC16] Mohammad Masudur Rahman, Chanchal K. Roy, and Jason A. Collins. CoRReCT: code reviewer recommendation in GitHub based

on cross-project and technology experience. In *ICSE (Companion Volume)*, pages 222–231. ACM, 2016.

[RRH+18] Juri Di Rocco, Davide Di Ruscio, Johannes Härtel, Ludovico Iovino, Ralf Lämmel, and Alfonso Pierantonio. Systematic Recovery of MDE Technology Usage. In *ICMT*, volume 10888 of *Lecture Notes in Computer Science*, pages 110–126. Springer, 2018.

[RRH+20] Juri Di Rocco, Davide Di Ruscio, Johannes Härtel, Ludovico Iovino, Ralf Lämmel, and Alfonso Pierantonio. Understanding MDE projects: megamodels to the rescue for architecture recovery. *Softw. Syst. Model.*, 19(2):401–423, 2020.

[RvDV12] Steven Raemaekers, Arie van Deursen, and Joost Visser. Measuring software library stability through historical version analysis. In *ICSM*, pages 378–387. IEEE Computer Society, 2012.

[RvDV13] Steven Raemaekers, Arie van Deursen, and Joost Visser. The maven repository dataset of metrics, changes, and dependencies. In *MSR*, pages 221–224. IEEE Computer Society, 2013.

[RvDV17] Steven Raemaekers, Arie van Deursen, and Joost Visser. Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software*, 129:140–158, 2017.

[SAB18] Davide Spadini, Maurício Finavaro Aniche, and Alberto Bacchelli. Pydriller: Python framework for mining software repositories. In *ESEC/SIGSOFT FSE*, pages 908–911. ACM, 2018.

[SAH10] Weiyi Shang, Bram Adams, and Ahmed E. Hassan. An experience report on scaling tools for mining software repositories using MapReduce. In *ASE*, pages 275–284. ACM, 2010.

[SB10] Yannis Smaragdakis and Martin Bravenboer. Using Datalog for Fast and Easy Program Analysis. In *Datalog*, volume 6702 of *Lecture Notes in Computer Science*, pages 245–251. Springer, 2010.

[SB15] Anand Ashok Sawant and Alberto Bacchelli. A Dataset for API Usage. In *MSR*, pages 506–509. IEEE Computer Society, 2015.

[SBC10] Philip Stutz, Abraham Bernstein, and William W. Cohen. Signal/-Collect: Graph Algorithms for the (Semantic) Web. In *International Semantic Web Conference (1)*, volume 6496 of *Lecture Notes in Computer Science*, pages 764–780. Springer, 2010.

[SBEV18] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. Incrementalizing lattice-based program analyses in Datalog. *Proc. ACM Program. Lang.*, 2(OOPSLA):139:1–139:29, 2018.

[SBH+19] César Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais, and Benoit Baudry. The emergence of software diversity in maven central. In *MSR*, pages 333–343. IEEE / ACM, 2019.

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[SCC02] William R Shadish, Thomas D Cook, and Donald T Campbell. *Experimental and quasi-experimental designs for generalized causal inference.* Houghton Mifflin Company, 2002.

[Sch06] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.

[SdLJPM18] Daricélio Moreira Soares, Manoel Limeira de Lima Júnior, Alexandre Plastino, and Leonardo Murta. What factors influence the reviewer assignment to pull requests? *Information & Software Technology*, 98:32–43, 2018.

[SDNR07] Warren Shen, AnHai Doan, Jeffrey F. Naughton, and Raghu Ramakrishnan. Declarative Information Extraction Using Datalog with Embedded Extraction Predicates. In *VLDB*, pages 1033–1044. ACM, 2007.

[SG18] Diomidis Spinellis and Georgios Gousios. How to analyze git repositories with command line tools: we're not in kansas anymore. In *ICSE (Companion Volume)*, pages 540–541. ACM, 2018.

[Sha93] Jun Shao. Linear model selection by cross-validation. *Journal of the American statistical Association*, 88(422):486–494, 1993.

[SHH+05] Dag I. K. Sjøberg, Jo Erskine Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanovic, Nils-Kristian Liborg, and Anette C. Rekdal. A Survey of Controlled Experiments in Software Engineering. *IEEE Trans. Software Eng.*, 31(9):733–753, 2005.

[SHL+19] Philipp Seifer, Johannes Härtel, Martin Leinberger, Ralf Lämmel, and Steffen Staab. Empirical study on the usage of graph query languages in open source Java projects. In *SLE*, pages 152–166. ACM, 2019.

[SJAH09] Weiyi Shang, Zhen Ming Jiang, Bram Adams, and Ahmed E. Hassan. MapReduce as a general framework to support research in Mining Software Repositories (MSR). In *MSR*, pages 21–30. IEEE Computer Society, 2009.

[SKP+20] Mohammed Sayagh, Noureddine Kerzazi, Fábio Petrillo, Khalil Bennani, and Bram Adams. What should your run-time configuration framework do to help developers? *Empir. Softw. Eng.*, 25(2):1259–1293, 2020.

[SL09] Taemin Seo and Heesang Lee. Agent-based Simulation Model for the Evolution Process of Open Source Software. In *SEKE*, pages 170–177. Knowledge Systems Institute Graduate School, 2009.

[SL16] Thomas Schmorleiz and Ralf Lämmel. Similarity management of 'cloned and owned' variants. In *SAC*, pages 1466–1471. ACM, 2016.

[SME+17] Anas Shatnawi, Hafedh Mili, Ghizlane El-Boussaidi, Anis Boubaker, Yann-Gaël Guéhéneuc, Naouel Moha, Jean Privat, and Manel Abdellatif. Analyzing program dependencies in Java EE applications. In *Proc. MSR*, 2017.

[SMS16] Ingo Scholtes, Pavlin Mavrodiev, and Frank Schweitzer. From Aristotle to Ringelmann: a large-scale analysis of team productivity and coordination in Open Source Software projects. *Empir. Softw. Eng.*, 21(2):642–683, 2016.

[SPN+18] Pasquale Salza, Fabio Palomba, Dario Di Nucci, Cosmo D'Uva, Andrea De Lucia, and Filomena Ferrucci. Do developers update third-party libraries in mobile apps? In *ICPC*, pages 255–265. ACM, 2018.

[SRN+14] Reinout Stevens, Coen De Roover, Carlos Noguera, Andy Kellens, and Viviane Jonckers. A logic foundation for a general-purpose history querying tool. *Sci. Comput. Program.*, 96:107–120, 2014.

[SSM18] Victoria Stodden, Jennifer Seiler, and Zhaokun Ma. An empirical analysis of journal policy effectiveness for computational reproducibility. *Proc. Natl. Acad. Sci. USA*, 115(11):2584–2589, 2018.

[Ste15] Reinout Stevens. A Declarative Foundation for Comprehensive History Querying. In *ICSE (2)*, pages 907–910. IEEE Computer Society, 2015.

[Sza21] Tamás Szabó. *Incrementalizing Static Analyses in Datalog*. PhD thesis, University of Mainz, Germany, 2021.

[SZZ05] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *MSR*. ACM, 2005.

[TBP+17] Michele Tufano, Gabriele Bavota, Denys Poshyvanyk, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. An empirical study on developer-related factors characterizing fix-inducing commits. *J. Softw. Evol. Process.*, 29(1), 2017.

[TDH14] Jason Tsay, Laura Dabbish, and James D. Herbsleb. Influence of social and technical factors for evaluating contribution in GitHub. In *ICSE*, pages 356–366. ACM, 2014.

[TH18] Chakkrit Tantithamthavorn and Ahmed E. Hassan. An experience report on defect modelling in practice: pitfalls and challenges. In *ICSE (SEIP)*, pages 286–295. ACM, 2018.

[TLPH95] Walter F. Tichy, Paul Lukowicz, Lutz Prechelt, and Ernst A. Heinz. Experimental evaluation in computer science: A quantitative study. *J. Syst. Softw.*, 28(1):9–18, 1995.

[TM03] Tom Tourwé and Tom Mens. Identifying Refactoring Opportunities Using Logic Meta Programmin. In *CSMR*, pages 91–100. IEEE Computer Society, 2003.

[TME+18] Nikolaos Tsantalis, Matin Mansouri, Laleh Mousavi Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *ICSE*, pages 483–494. ACM, 2018.

[TMHI16] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *ICSE*, pages 1039–1050. ACM, 2016.

[TMHM17] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. *IEEE Trans. Software Eng.*, 43(1):1–18, 2017.

[TTDM15] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. Online Defect Prediction for Imbalanced Data. In *ICSE (2)*, pages 99–108. IEEE Computer Society, 2015.

[VCR22] Camilo Velázquez-Rodríguez, Eleni Constantinou, and Coen De Roover. Uncovering Library Features from API Usage on Stack Overflow. In *SANER*, pages 207–217. IEEE, 2022.

[Voh16] Deepak Vohra. *Apache Parquet*, pages 325–335. Apress, Berkeley, CA, 2016.

[Vok04] Marek Vokác. Defect Frequency and Design Patterns: An Empirical Study of Industrial Code. *IEEE Trans. Software Eng.*, 30(12):904–917, 2004.

[VPR+15] Bogdan Vasilescu, Daryl Posnett, Baishakhi Ray, Mark G. J. van den Brand, Alexander Serebrenik, Premkumar T. Devanbu, and Vladimir Filkov. Gender and Tenure Diversity in GitHub Teams. In *CHI*, pages 3789–3798. ACM, 2015.

[VSF15] Bogdan Vasilescu, Alexander Serebrenik, and Vladimir Filkov. A Data Set for Social Diversity Studies of GitHub Teams. In *MSR*, pages 514–517. IEEE Computer Society, 2015.

[WL14] Simon Weber and Jiebo Luo. What Makes an Open Source Code Popular on Git Hub? In *ICDM Workshops*, pages 851–855. IEEE Computer Society, 2014.

[YMNC04] Annie T. T. Ying, Gail C. Murphy, Raymond T. Ng, and Mark Chu-Carroll. Predicting Source Code Changes by Mining Change History. *IEEE Trans. Software Eng.*, 30(9):574–586, 2004.

[YWF+15] Yue Yu, Huaimin Wang, Vladimir Filkov, Premkumar T. Devanbu, and Bogdan Vasilescu. Wait for It: Determinants of Pull Request Evaluation Latency on GitHub. In *MSR*, pages 367–371. IEEE Computer Society, 2015.

[YXF+20] Meng Yan, Xin Xia, Yuanrui Fan, David Lo, Ahmed E. Hassan, and Xindong Zhang. Effort-aware just-in-time defect identification in practice: a case study at Alibaba. In *ESEC/SIGSOFT FSE*, pages 1308–1319. ACM, 2020.

[YYY+12] Cairong Yan, Xin Yang, Ze Yu, Min Li, and Xiaolin Li. IncMR: Incremental Data Processing Based on MapReduce. In *IEEE CLOUD*, pages 534–541. IEEE Computer Society, 2012.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, pages 15–28. USENIX Association, 2012.

[ZHMZ17] Feng Zhang, Ahmed E. Hassan, Shane McIntosh, and Ying Zou. The Use of Summation to Aggregate Software Metrics Hinders the Performance of Defect Prediction Models. *IEEE Trans. Software Eng.*, 43(5):476–491, 2017.

[ZLH+21] Kaisheng Zeng, Chengjiang Li, Lei Hou, Juanzi Li, and Ling Feng. A comprehensive survey of entity alignment for knowledge graphs. *AI Open*, 2:1–13, 2021.

[ZMDR22] Ahmed Zerouali, Tom Mens, Alexandre Decan, and Coen De Roover. On the impact of security vulnerabilities in the npm and RubyGems dependency networks. *Empir. Softw. Eng.*, 27(5):107, 2022.

[ZN08] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *ICSE*, pages 531–540. ACM, 2008.

[ZPZ07] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *PROMISE 2007*, page 76. IEEE, 2007.

[ZS06] Yuefeng Zhang and Dhaval Sheth. Mining Software Repositories for Model-Driven Development. *IEEE Softw.*, 23(1):82–90, 2006.