Understanding and Supporting Software Model Evolution through Edit Operation Mining and AI-based Software Model Completion

A dissertation submitted towards the degree
Doctor of Engineering (Dr.-Ing.)
of the Faculty of Mathematics and Computer Science
of Saarland University

by

Christof Tinnes

Saarbrücken, 2025

UNIVERSITÄT
DES
SAARLANDES

Dean of the Faculty:	Prof. Dr. Roland Speicher
Day of the Colloquium:	October 24th, 2025
Examination Board:	
Chair of the Defense:	Prof. Dr. Isabel Valera
Reviewers:	Prof. Dr. Sven Apel
	Prof. Dr. Marianne Huchard
	Prof. Dr. Manuel Wimmer
Academic Assistant:	Dr. Marvin Wyrich

Preface

My journey of science began with a fascination for dynamical systems and the question "How do systems evolve?". For my undergraduate studies, I gravitated towards mathematics and mathematical physics, where empirical observations in physics are often elegantly distilled into mathematical descriptions of systems. For example, Newton's laws of motion together with Newton's law of universal gravitation, in theory, approximately describe the motion of objects under gravitational influence. Before Newton, astronomers such as Tycho Brahe, Johannes Kepler, and Nikolaus Kopernikus, have collected tons of data about the movements of planets, and tried to come up with manageable descriptions of the motions, such as Kepler's laws of planetary motion.

This inevitably leads to the question of what "makes a good description of the observations" of a dynamical system. Of course, on the one hand, this description should be able to describe the observations themselves to a certain degree of accuracy. On the other hand, as new observations are being made, these observations should not contradict the description of the system. That is, the description should have predictive capabilities.

This led me to the realization that the major questing driving my curiosity is: "How are the past and future interconnected?" This question makes use of the concept of time and the idea of an arrow of time.

But why would one actually want a description of the past with predictive capabilities? Even though I have an intrinsic motivation in understanding the behavior of systems, another reason is that we—as humanity—want knowledge to improve our decision-making. To stick with the example of gravitational systems, a good predictive model is a key ingredient to launch rockets and bring satellites into orbit.

Unfortunately, as it turns out, if the system's description has non-linearities, already simple systems do not have a closed-form solution. Already for three bodies—point masses, to be more precise—there is no general closed-form solution to the equations of motion, under Newtonian gravitation, to predict their movement. Still, using concepts from perturbation theory and control theory, we can use the models for decision-making—even managing to dock a spacecraft to a space station or an asteroid.

One thing always puzzled me in the analysis of dynamical systems: According to Poincaré's Recurrence Theorem, every trajectory of dynamical system, fulfilling several criteria such as phase space volume preservation, will eventually return to an arbitrarily small neighborhood of its starting point. Even though this time will often be astronomically large, it is still finite, which completely contradicts our intuition of time. If there is a "closed universe" following the laws of physics how we describe them, then the universe will eventually return to its initial state.

After finishing my masters in mathematical physics, I turned towards software engineering, where I found a parallel: software, like systems, evolves rather chaotically.

Indeed, a runtime software systems computes a computable function. During the software development lifecycle, this function continuously changes, giving rise to a dynamical system on the space of computable functions. Software development methodologies, indeed, arose from the need to have a control loop between the environment—described by the problem space—and the runtime system—described by the solution space. From a dynamical systems point of view, software engineering is not well understood. Most of the research has been focusing on the question of how to engineer software, but not on the mathematical description of software evolution and the adaptation process of software to its environment.

In the past, in this conflict of perpetual change, we surrender to the unpredictability, acknowledging that no "world formula" can encapsulate it all and predict the future state of a system. It is exactly this non-existence of a world formula that forces us to build models from our incomplete knowledge, which inevitably leads to the necessity of evolution: To be able to make any decisions, we are forced to separate the whole “world” into a system under concern and its environment. Only if the subsystem nicely fits into the environment, it will become a building block of future systems. This massively constraints the whole world, because not every subsystem will nicely fit into the environment. This shows that we can not expect ergodicity, and more generally, it is unlikely that a world formula would be volume preserving on the space of system states. Therefore, Poincaré’s Recurrence Theorem does not necessarily hold. Conversely, if we believe Poincaré’s Recurrence Theorem does not hold for a theoretic world formula that predicts the future, the world is either shrinking—which we can rule out because we observe increasing complexities—, growing or not constraint via a closed boundary at all. What is remarkable about this thought is that the decomposition of the whole into a system under concern and its environment—drawing a boundary around a system—is an arbitrary act of the observer, a thought related to the concept of downward causation.

Our adaptation to changing environments tremendously narrows down conceivable designs, ensuring predictability in the short run. Paradigm shifts, however, occur like sudden phase transitions—unforeseeable yet transformative and often lead to the “creative destruction” of previous paradigms.

Even though the prediction of sudden phase transitions is currently out of reach in the realm of software, a better predictability in the short run is a desirable goal. This reinforces the leading question behind this thesis: “How are the past and future interconnected and what is "the best model" for predicting the future?” This question will always be accompanied by the question of what leads to the arrow of time.

It has long been hypothesized that a good explanation (or hypothesis) for an observation is the simplest one that explains the observation. But how would this principle apply to software engineering, how can it be made practical, and why should this even hold true? We will see that a more practical formulation of this principle is the Minimum Description Length principle, which basically states that the most compressing model is the best model.

Generative models that recently gained popularity to a larger audience, are a related concept with many practically usable implementations. Generative machine learning models compress huge amounts of data into a comparatively small set of parameters and are able to generate new data or continue an unseen data sequence.

Regarding the question of the arrow of time, from an engineering perspective, this is quite simple: new building blocks comprise older building blocks, but not vice versa.

The path to this thesis itself, on a meta-level, is a journey of compression and reuse. The theory developed to better understand the driving question of the connection of the past and future build on top of the work of others—such as Kolmogorov complexity, the Minimum Description Length principle, or the concepts of software model transformations or generative machine learning models.

Ultimately, this work represents not only a scholarly endeavor but a personal journey infused with curiosity, reflection, and a passion to unravel the ties binding the past to the future in the tapestry of software evolution.

Abstract

Model-based Systems Engineering has become increasingly important in managing the complexity of modern software systems. However, the evolution of software models in large-scale, real-world projects remains a significant challenge due to the lack of effective and automated methods. This thesis addresses this problem by providing a comprehensive theoretical foundation for software model evolution, and, based on this theory, presents practical approaches to understand and support model evolution, which will be evaluated and validated by empirical studies. The first part of the thesis develops a theory to software model evolution and introduces novel techniques—backing the theory—based on graph mining, large language models, and graph neural networks. These methods allow to automatically define model transformations, and support software model completion, solely based on model histories from model version control systems. The second part motivates and shows the relevance and usefulness of the research behind the thesis by providing insights into the complexity of a real-world industrial model-driven product line at our industry partner. In the third part, controlled experiments provide evidence for the theory developed in the first part of the thesis. The findings demonstrate the feasibility and that there is a potential for data-driven intelligent modeling assistants to support model-based engineering—not only boosting productivity of engineers, but also accuracy and overall quality of software models. This research contributes to advancing Model-based Systems Engineering by proposing automated solutions that support the continuous evolution of software models, ultimately leading to more robust and maintainable systems.

Zusammenfassung

Modellbasiertes Systems Engineering hat sich bei der Bewältigung der Komplexität moderner Softwaresysteme als zunehmend wichtig erwiesen. Die Evolution von Softwaremodellen in großen, realen Projekten bleibt jedoch aufgrund des Mangels an effektiven und automatisierten Methoden eine erhebliche Herausforderung. Diese Arbeit befasst sich mit diesem Problem, indem sie eine umfassende theoretische Grundlage für die Evolution von Softwaremodellen bereitstellt und darauf aufbauend praktische Ansätze zum Verständnis und zur Unterstützung der Modellevolution vorstellt, die durch empirische Studien evaluiert und validiert werden.

Der erste Teil der Arbeit entwickelt eine Theorie zur Evolution von Softwaremodellen und führt neuartige Techniken — zur Untermauerung der Theorie -– ein, die auf Graph Mining, großen Sprachmodellen und Graph Neural Networks basieren. Diese Methoden ermöglichen es, Modelltransformationen automatisch zu definieren und die Vervollständigung von Softwaremodellen zu unterstützen, ausschließlich basierend auf Modellhistorien aus Modellversionskontrollsystemen.

Der zweite Teil motiviert und zeigt die Relevanz und Nützlichkeit der Forschung hinter dieser Arbeit, indem er Einblicke in die Komplexität einer realen, modellgetriebenen Produktlinie bei unserem Industriepartner gibt.

Im dritten Teil liefern kontrollierte Experimente Belege für die im ersten Teil der Arbeit entwickelte Theorie. Die Ergebnisse demonstrieren die Machbarkeit und das Potenzial datengestützter intelligenter Modellierungsassistenten zur Unterstützung des modellbasierten Engineerings – nicht nur zur Steigerung der Produktivität der Ingenieure, sondern auch der Genauigkeit und der Gesamtqualität von Softwaremodellen. Diese Forschung trägt dazu bei, das modellbasierte Systems Engineering voranzutreiben, indem sie automatisierte Lösungen vorschlägt, die die kontinuierliche Evolution von Softwaremodellen unterstützen und letztendlich zu robusteren und wartbareren Systemen führen.

Acknowledgments

First and foremost, my deepest gratitude goes to my friends and family, especially my girlfriend Kristina and my daughter Milena, for their unwavering love and support. Their patience during my late-night and weekend writing sessions, and their constant encouragement, have been invaluable.

At the software engineering chair of Saarland University, there are many who I have to thank: First, my supervisor, Prof. Dr. Sven Apel has been an exceptional advisor and mentor, introducing me to the dynamic world of software engineering research. I am particularly grateful to Alisa Welter for her remarkable master thesis, which significantly contributed to my work, and her relentless efforts in enhancing and collaborating on the publication regarding software model completion. Dr. Florian Sattler offered crucial feedback on this thesis, while Kallistos Weis and I shared many insightful hallway conversations and enlightening hikes, and all others in the group who provided feedback and support.

At my employer, Siemens, I received immense support from my colleagues, notably Dr. Uwe Hohenstein, whose guidance was pivotal, especially at the outset of my studies. Prof. Dr. Andreas Biesdorf challenged my thinking and connected me with relevant industry contacts, leading to the creation of the thesis's industrial dataset. Martin Kramer and I engaged in numerous forward-thinking discussions about AI assistant systems in software architecture long before they gained popularity. Dr. Manoj Mahabaleshwar, whose journey preceded mine, offered incisive and targeted feedback on a technical level. Alfred Feldmeyer served as a sparring partner, testing ideas and solutions while providing valuable insights. My manager, Denis Schroff, consistently displayed support, openness, and a willingness to embrace new ideas. I am equally thankful to my team and other colleagues at Siemens who supported me in various ways.

Regarding academia, Prof. Dr. Prem Devanbu provided critical insights into transferring ideas from artificial intelligence in the coding domain to the model-driven engineering domain. Prof. Dr. Jilles Vreeken offered pointers and feedback during my qualifying exam that provided valuable guidance for the future trajectory of my research. Prof. Dr. Timo Kehrer contributed ideas on the pertinence of edit operations, utilizing his skill to simplify overly complex texts while maintaining a constructive approach.

I owe special thanks to Prof. Dr. Thomas Fuchss for his mentorship, thought-provoking discussions, and valuable feedback on writing and structuring thoughts.

Dr. Maik Reddinger, an admired colleague from my master's studies with exceptional skills in the mathematical modeling of ideas, has been a reliable resource for theoretical challenges. Dr. Mitchell Joblin enriched my work with his collaboration and feedback on edit operation mining and graph neural networks.

At the Sebis Chair at the Technical University of Munich, I extend my appreciation to my room neighbor, Dr. Dominik Huth, for his timely and helpful ideas, as well as to Prof. Dr. Florian Matthes, and the many others whose support was instrumental to this thesis.

Contents

List of Figures

List of Tables

Code Listings

1 Introduction

To do science is to search for repeated patterns, not simply to accumulate facts.
— Robert MacArthur

1.1 Motivation

1.1.1 Software Evolution

In 1936, Alan Turing proposed the idea of a universal machine that could simulate any other machine thereby laying the foundation for software as we know it today [267, 327]. Alan Turing's idea, with the program entirely held in electronic memory, was first realized in 1948 at the University of Manchester [267], marking the first appearance of software. Since this inaugural appearance software and software ecosystems have been subject to evolution [200, 201, 220, 221]—similar to natural structures such as organisms. Unlike natural evolution, software typically evolves through a design process (i.e., human effort) rather than through autonomous reproduction processes.[1] Through the principle of selection [76], software must adapt to its environment, which is given by the sociotechnical system [31] in which it is embedded—comprising the software system, other software systems, hardware, as well as human stakeholders and organizational processes it interacts with. Consequently, software has to co-evolve with the entire sociotechnical system.[2] As an

1 Darwin's evolution theory, of course, cannot be directly applied to the evolution of software. Other theories, for example, from the study of cultural evolution or theories such as Assembly Theory [295], would be more appropriate frameworks for describing and studying software evolution. The comparison with Darwin's evolution theory—a comparison often drawn in the literature [113]—is made here for the sake of simplicity, and some high-level key principles of evolution theory do apply to sociotechnical systems [333], indeed. Due to the entanglement between sociocultural and technical mutual adaptation, the picture is more complex for software evolution, and principles such as selection can even be overridden, for example, by political intervention.

2 For example, software functionalities for barrier-free access to trains of the Deutsche Bahn have been required by German regulation https://www.gesetze-im-internet.de/bgg, which itself is the consequence of a long history of the topic of equal participation of people with disabilities in society.

example for this coupling or embedding, for the end-user of a software system, the perceived *value-add* needs to outweigh the price that the user has to pay for the software. Therefore, for the software to meet a product-market fit and become profitable in the long run, the perceived value-add needs to be higher than the total costs of the software.

Software evolution involves adapting the system to changing requirements, environments, dependencies, and technologies [201, 221, 225, 328]. A single software system (i.e., an individual in the software ecosystem) typically undergoes a kind of *aging process* [201, 251] during its continuous adaptation to the environment, for example, through incorporating new features, bug fixes, or groomative changes. As for natural organisms, this often leads to an increasing complexity [200, 201].[3]

1.1.2 Coping with Complexity and Model-based Engineering

The ever-increasing complexity of software has been linked to a decreasing quality, software projects running over budget and projects running over time, a process that led to the so-called *software crisis* [238]. In 1968, this software crisis has been the central topic of the first NATO Software Engineering Conference [238], sometimes considered as origin of the field of *software engineering* [348].

Since then, several techniques and methodologies to manage the complexity of software systems have been developed [93, 192, 272, 289]. For example, to ease the management of, communication about, and automation of several aspects of the software system, documenting and modeling the systems themselves have therefore become increasingly important [101, 105, 187, 283]. This development led to the creation of software development methodologies that focused more around software models themselves. In *Model-driven Engineering* [169], models of the software system become primary artifacts[4], meaning they are the central artifacts of the *software development lifecycle*. Unfortunately, explicitly modeling and documenting software and software systems is typically considered a cumbersome activity, and the willingness to spend time on documentation efforts are typically low [202, 252]. Software documentation and modeling often requires dealing with complex and heterogeneous artifacts that capture different aspects of the system under development [45]. Modeling tools have been developed to support users managing this complexity. Besides

3 "Objects" are usually build out of existing "objects" and one can observe also a tendency to use more recent "objects" over "older objects". For example, one would build a new software system rather from modern libraries and programming languages than from machine code. Support for this assertion also comes from several studies [4, 54, 125, 295]. For example, Assembly Theory [125, 295] introduces a quantity called *assembly index* and shows that the assembly index for living and life-derived objects is typically higher than for abiotic objects. In the course of this thesis, we will discuss this assertion in greater detail. Note that, as we will see below, the perceived complexity can indeed be reduced by abstraction mechanisms and information hiding.

4 For this thesis, everything that can be used to communicate about the software system as part of the software development lifecycle—including source code—is considered an artifact.

support for editing software models in textual or graphical notation, these tools often offer support for: (1) *model transformations* [224]—transforming models from one notation or abstraction level to another, such as from UML to code, or from code to UML; (2) *model execution* [73, 347]—to simulate or test the model's behavior; (3) *model analysis* [20, 233]—to analyze models for various purposes, such as verifying, validating, optimizing, or evaluating models; and (4) *model management* [39, 130]—to manage model artifacts and their relationships, such as versioning, configuration, traceability, and enabling collaborative editing of these artifacts.

1.1.3 Challenges of Modeling Tools

Despite a pressing need, currently available support for the evolution of models in modeling tools such as Eclipse EMF, MagicDraw, Simulink, Papyrus, or Rhapsody is very limited [159]. In fact, besides graphical modeling functionalities and support for several modeling and domain-specific languages, tool support is typically limited to version control [185, 290], model differencing, model merging capabilities [17, 182, 313], and only rudimentary support for model refactorings [222] and other model transformations. Often, no mature or appropriate tooling is available for the task at hand, which hinders adoption of the concepts in Model-driven Engineering [45, 50, 105, 139]. It has been even argued in the literature [21, 50, 105] that (perceived) efforts and the costs of modeling activities may even outweigh the advantages. Functionalities such as change propagation [168], model completion [50, 236, 292, 311], model generation [78, 294], model repair [243, 244], model explanation [50], and features to better understand models and model evolution [161] are rather subject to research than being used in state-of-the-art modeling tools.

1.1.4 Intelligent Modeling Assistance: The Future for Software Modeling?

To support modeling tool users in their often complex tasks, it has been proposed to improve tool support—in particular, for manipulating existing software models [71, 105]. Cabot et al., for example, argue that the perceived value-add needs to outweigh the cost of modeling to foster the adoption of modeling [50]. They envision to leverage existing knowledge inside the modeling tools. This leveraging of knowledge (also called *cognification* [50]) is not limited to deep learning but also includes leveraging existing human intelligence (e.g., in crowdsourcing). Intelligent tool support could provide modelers with guidance, suggestions, and automation—in particular, for their evolution tasks [50, 71, 235]—thereby increasing the value-add, and therefore also the return on investment that a modeling tool provides to the user.

In this thesis, we will follow this vision of cognifying Model-driven Engineering. In particular, we will investigate several use cases and aspects of *intelligent modeling*

assistance. We will focus on use cases related to the evolution of models, where we learn from the history of software models to support future modeling activities. In particular, we will investigate the basic building blocks of model evolution, the so-called *edit operations*, and how they can be identified in the modeling history—usually available in the form of model repositories. Edit operations can then be used for many tasks, for example, to apply them to models to modify them (e.g., adding functionality), or for semantic lifting (see Section 6.3) to better understand model differences (i.e., how models evolve). However, given as set of edit operations, the modeler has to choose the right edit operations and configure them to adapt them to the concrete context by setting attributes and defining references to other model elements. This is a repetitive and error-prone task, and some attributes are already clear from the context—for example, a model element's name might be derived by the name of the model elements surrounding it. Thus, we will investigate model (auto)-completion to understand whether—and to what extent—available technology can be leveraged to support the modeler in the evolution of software models in the context of Intelligent Modeling Assistance.

The guiding vision of this thesis is that intelligent modeling assistance can foster the adoption of modeling tools and Model-driven Engineering, in general [50].

Vision for Intelligent Modeling Assistants: *Our vision is to foster the adoption of model-based software engineering by providing intelligent tool support in the form of Intelligent Modeling Assistants that leverage existing knowledge to optimally support key stakeholders along the development lifecycle.*

By "key stakeholders", we refer to modelers, developers, architects, and domain experts. The concrete stakeholders depend on the modeling language and the modeling domain. This thesis focuses on software and system models, therefore the key stakeholders are software and system engineers and architects.

To make this vision more tangible, we envision tool support, similar to GitHub Copilot for textual artifacts—and source code, in particular—in the form of a copilot that is directly integrated with the modeling tool, as illustrated in Figure 1.1.

As we want to leverage *existing* knowledge, the vision is based on the following working hypothesis:

Hypothesis

Hypothesis 1 (Working Hypothesis)**.** There is a vast amount of knowledge (e.g., in the form of model repositories, other artifacts, implicit knowledge of stakeholders) that can be leveraged to provide intelligent tool support to the stakeholders. Some enabled tool support will be perceived useful[a] by the stakeholders.

a In fact, for adoption the perceived value-add needs to outweigh the costs in the long-run.

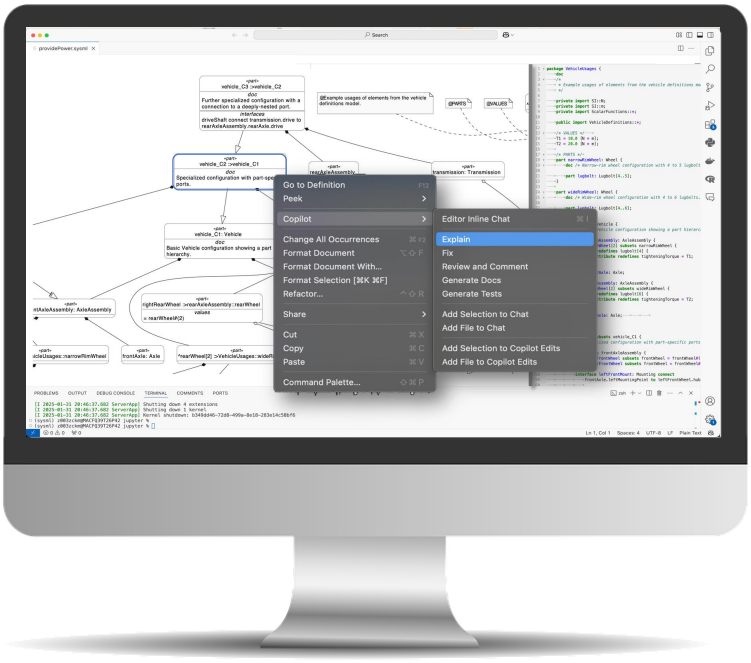

Figure 1.1: A mockup of a SysML CoPilot in action, similar to GitHub CoPilot. The copilot provides chat, explain, fix, review and comment, generate docs, and generate test functionalities via a context menu.

1.2 Research Goals

In this thesis, we contribute to realizing this vision of an intelligent modeling assistant—focusing on aspects of model evolution (in the product and time dimension) such as understanding as well as supporting model evolution. The core of this thesis investigates *edit operations* and how they can be identified in the modeling history—usually available in the form of model repositories. Edit operations, roughly speaking, are *transformations* applied to software models to evolve them over time, typically involving simple changes such as adding, removing, or modifying model elements, or sometimes more complex changes to the models. That is, edit operations are the basic operations offered by the modeling tool to the user to manipulate the model (mostly by GUI operations, but also via queries and scripts). These edit operations can simplify and automate many tasks that involve model changes, such as repair [127, 244], refactoring [20, 234], co-evolution [19, 111, 132, 184, 270], model generation [258], and model completion [180, 236, 311]. However, specifying edit operations for different domains and tasks is challenging and requires expert knowledge and effort. We therefore approach the issue from a data-driven perspective and investigate whether this expert knowledge can also be derived from model histories. We explore classic approaches such as *frequent subgraph mining*, but also more recent *deep-learning* approaches for this purpose. For the concrete use case of *model (auto-)completion*, we investigate whether we can even circumvent the

explicit knowledge of edit operations for the model completion and directly learn how to complete software models from the modeling history.

More generally, the key problem addressed in this thesis is the lack of effective and automated methods for understanding and supporting the evolution of software models in *Model-based* and *Model-driven Software Engineering* environments—in particular, understanding changes of models in time and across different software products, and supporting model evolution by automated model completion.

The comparison and merging of disparate software models, the detection and resolution of inconsistencies, and correctly editing the current model state to achieve a certain goal represent complex tasks that necessitate considerable manual effort and expertise. State-of-the-art modeling tool support frequently relies on manual interventions and reviews, which are time-consuming and susceptible to errors. Motivated by the success of modern integrated development environments (IDEs) for source code, there is a clear desire for intelligent modeling assistants that support software engineers in these tasks. Failure to address these challenges increases reluctance to pursue a model-based or model-driven development paradigm, given that the upfront investments for model-based development are not outweighed by its benefits. Still, the benefits of model-based development are manifold, including eased communication about the system under development and increasing software quality [45, 105]. In particular in safety critical systems, such as train control software, the benefits of model-based development are even more pronounced, as the quality of the software is of utmost importance, and requirements traceability [320] is often a key concern or even a legal requirement. Especially in these domains, modeling tools are heavily used, although, as we will see, several challenges related to tool support need to be addressed. Additionally, improving tool support for model evolution can also foster the adoption of model-based development in other domains.

This thesis aims to, first, develop a theory of software model evolution and, second, to explore and develop novel techniques using *graph mining*, *large language models*, and *graph neural networks* to better understand the merits of different basic technologies for key tasks in software model evolution. Current research in this area has predominantly focused on algorithmic, *rule-based approaches*, which are often limited in their ability to generalize across different software models and application domains. The rationale for this endeavor is twofold. Firstly, there is a wealth of engineering data available in open-source repositories, as well as in company-owned private repositories. These repositories ease the collaboration between different modelers and stakeholders and provide a version history of the models. Secondly, there is a need to investigate the potential of *data-driven approaches* to bring these valuable data resources to light. By addressing this problem, this dissertation seeks to provide knowledge towards further enhancing the productivity, increasing the accuracy, reducing maintenance burden, and improving the overall quality of Model-based and Model-driven Software Engineering.

Our motivation for this research originates from a real-world industrial software product line for train control software, where the evolution of software models has become a challenging endeavor. Even though motivated by this real-world industrial project this thesis delves into the challenges of model evolution independent of the concrete meta-model—increasing the generalizability[5] of the results. Indeed, the real-world case study was the starting point of this research but, as we will see later, the application context[6] of our research is not limited to this case study.
In summary, the goals of this thesis—guided be the vision for Intelligent Modeling Assistants—are to advance software model evolution research by

- developing a theory relating software model evolution to software model histories, in Part I of the thesis, (**Goal 1**)
- curating a diverse and substantial model evolution dataset that includes real-world industrial models, open-source models, and simulated models, in Part II of the thesis, (**Goal 2**)
- investigating the feasibility of data-driven methods for model evolution use cases—including edit operation mining, semantic lifting of model differences, and model completion, and evaluating these methods on the aforementioned model evolution dataset, in Part III of this thesis. (**Goal 3**)

Furthermore, this thesis strives to advance Model-driven Engineering itself, by proposing *concrete approaches* to understanding and supporting the evolution of models in Model-driven Engineering—in particular, approaches for edit operation mining and model completion.

1.3 Research Methodology

In this thesis, we pursue a mixed method empirical research [349] in the field of Model-driven Engineering.

In the first part of the thesis, we perform *theoretical research* and devise a *theory* that brings software evolution, pattern mining, machine learning, and generative models on a common ground. The theory is derived by a transfer and adaptation of existing ideas to the field of software evolution and Model-driven Engineering.

In the second part, we analyze a concrete real-world industrial project utilizing Model-driven Engineering. The analysis of this project is rather *qualitative case study*

5 It is also important to note that results should always be validated for a new application context. Also, specifics of the application context typically carry potential for an improvement over a general solution.

6 Our current dataset consists of the real-world case study, another dataset from a large amount of ECore models, and a synthetic dataset.

research with human subjects and serves as a starting point and motivation for the rest of our research contributions.

The third part of the thesis is mainly *empirical research* and investigates concrete use cases. We dig deeper into several more concrete Model-driven Engineering use cases, namely, *edit operation mining, semantic lifting of model differences, model evolution understanding in general, and model completion*. For these use cases, we propose concrete approaches and conduct mainly *quantitative research* to evaluate these approaches on a curated dataset of model histories. Statistical methods are used to analyze the experimental data (mainly collected via controlled experiments) and test clearly formulated hypotheses. This part can be considered as first empirical evidence for the theory derived in the first part of the thesis, although more research is needed to validate the theory.

Furthermore, literature reviews are performed qualitatively, for example we discuss pros and cons of existing approaches qualitatively, due to the lack of benchmarkable approaches in the research literature.

In principle, there are two approaches to this thesis:

The approach for the engineering researcher: From a software engineering perspective, concrete approaches for tool support in the area of model-based software development are proposed. Part 1 of the thesis provides a background and common basis for these approaches.

The approach of the scientific researcher: For the scientist, Part 1 (especially Section 3.5) can be understood as a theory for software evolution. Part 3 provides an initial evaluation of the theory by examining whether the theory can make meaningful predictions and works in reality.

1.4 Positioning of the Thesis in the Field of Software and System Evolution Research

On a higher level, this thesis investigates questions in the broader field of software and system engineering. More specifically the main questions of this thesis concerns the *evolution of software and systems* [33, 201, 221, 225, 328]. Especially, we focus on the evolution of models in the context of Model-driven Engineering. Evolution can occur in two dimensions: the product dimension and the time dimension [33], and we will touch on both. Model evolution, still, is a huge field, and we will dive into greater detail in two subfields: First, we report on a concrete real-world industrial project that makes use of Model-driven Engineering and discuss the merits intelligent modeling assistance [235] could bring in this setting. The research effort in this direction can be positioned as a case study research in the intersection of Model-driven Engineering

research and Software Product-Line Engineering research. Second, we investigate several use cases that could be considered as modeling recommender systems [14, 179]. We consider modeling recommender systems to be a subclass of Intelligent Modeling Assistants [235]. Many approaches in Model-driven Engineering research focus on rule-based or symbolic solutions (i.e., knowledge-based and content-based recommender systems), which rely on predefined rules or patterns to manipulate or analyze models. We will focus on solution approaches that we collectively refer to as *data-driven methods*. These approaches have in common that they leverage existing data to pursue their goal. On the one hand, given the example of edit operation mining, we show how explicit knowledge can be derived from existing data. On the other hand, given the example of model completion, we explore methods that directly exploit existing data for the task at hand.

This positioning of our research is depicted in Figure 1.2.

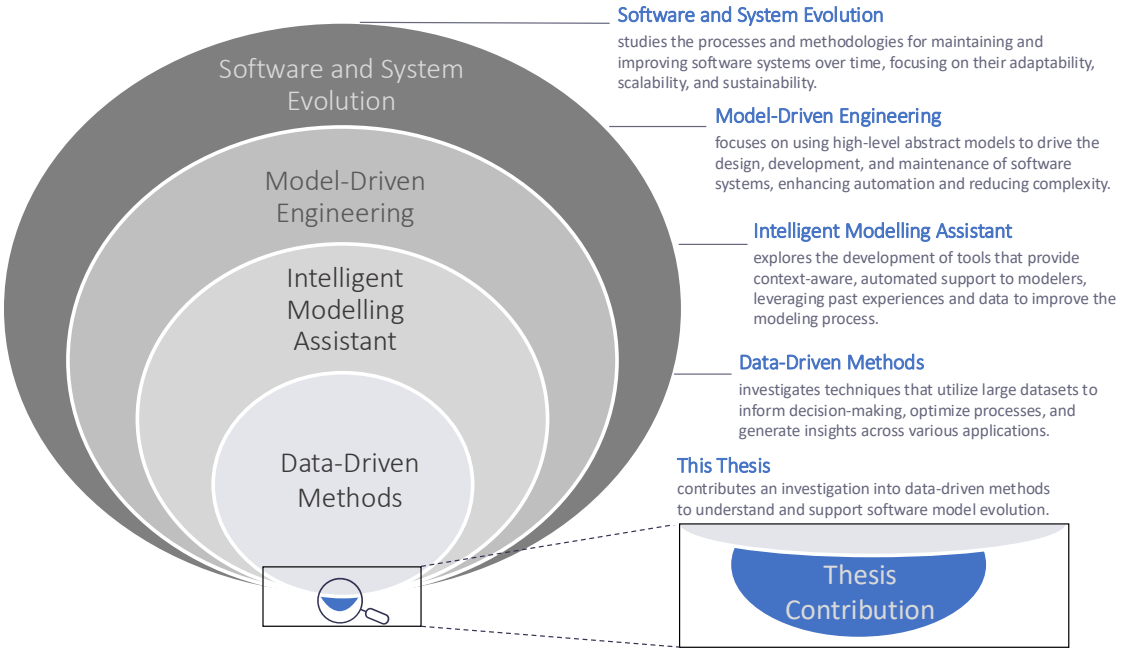

Figure 1.2: We position the thesis in the field of data-driven methods in Model-driven Engineering research (mainly empirical research).

1.5 Contributions and Key Results

With the research in this thesis we aim to advance the field of Model-driven Engineering research and to contribute to the field of Model-driven Engineering itself. Our contributions to the field of Model-driven Engineering research are:

An Evolution Dataset Towards a Model Evolution Benchmark. We present a curated model evolution dataset that includes real-world industrial models, open-

source models, and simulated models. With this dataset, we strive for a good trade-off between internal and external validity of our research conclusions. In particular, the simulated dataset gives us control over several parameters and therefore increases internal validity, while the real-world industrial and open-source datasets increase external validity. Given the example of model completion, from our experiments and analysis of existing approaches, we discuss a way to make approaches benchmarkable and thereby a way to overcome the reproducibility crisis. We demonstrate how to adapt an existing approach in a way to make it comparable to our approaches to model completion. The results of this analysis constitute a first step towards a benchmark in the field of model completion and intelligent modeling assistance, in general.

A New Real-World Case Study. We report on a concrete real-world, large-scale, industrial project which makes use of Model-driven Engineering and can be used to motivate our present research, but also future research in the field. This case study represents a complex and real-world application of Model-based Engineering and demonstrates the conditions to be found in real-world settings and therefore the (theoretical) requirements to be imposed on any proposed solution claiming real-world applicability. In this case study, we find that tool challenges are a major source of costs and that intelligent modeling assistance could be a solution to these challenges.

A Theory of Model Evolution. We will discuss a theory of model evolution. The idea is that we can learn from past evolution for future evolution. We will derive a theoretical solution to this problem and then discuss how to implement this solution in practice and what technologies are available for this implementation today.

Regarding our technical contributions to the field of Model-driven Engineering, we propose and investigate data-driven methods for model evolution use cases. These use cases are:

Edit Operation Mining. From given modeling histories, we contribute methods to identify edit operations as reoccurring edit patterns in the data. Edit operations can then be used for many other modeling activities, for example, model refactoring. We propose a concrete approach to edit operation mining and evaluate this approach on our model evolution dataset. We find that, indeed, the approach is able to identify meaningful edit operations in the data. This approach has limitations, though—such as scalability and no means of capturing semantics below type-level—and we discuss these limitations in detail.

Semantic Lifting of Model Differences. Differences between models are typically computed on a model element level and therefore fine-grained. These differences can be lifted to a higher level of abstraction, for example, to support

understanding of model differences. In combination with edit operation mining, we evaluate an earlier approach [161] in the context of our real-world case study and find that the approach is indeed able to lift model differences to a higher level and compress the information in the model differences.

Change Profiling. By aggregating model differences, we can create a change profile (i.e., a distribution of the edit operations in the model difference). These change profiles can then help to analyze and understand the evolution of a model. We investigate change profiles in the context of the real-world case study and find that change profiles can help to understand the evolution of a model, for example, by classifying model differences into different categories, such as, changes to documentation, functional modifications, or refactorings.

Model Completion. Model Completion is the task of suggesting model elements based on a given context. We investigate how we can learn to complete models based on data from the modeling history. We propose concrete approaches and evaluate these approaches on our model evolution dataset. Our approaches provide contextualized suggestions for model completion (down to an attribute-level), and we find that more than 62.30% of the suggestions are semantically correct in the context of the real-world case study.

Feature Extraction. When following an extractive approach to software product-line engineering or during reverse engineering a software product line from a set of products, one task is to identify, extract, and located so-called *features*. We investigate whether feature extraction approaches from the literature [214] can help to extract features from a given set of products in a real-world model-based software product line. We find that the class of approaches explored by us is not helpful in practice, mainly due to *noise* in the data.

The first four approaches are based on the theory of model evolution and are therefore evaluated on the model evolution dataset. If we apply the theory to variability "in space" (i.e., to several co-existing products) instead of to variability in time, we arrive at the feature extraction approach. This is also our main reason to include the feature extraction approach in this thesis.

For the use cases that we investigate, there are *downstream* activities that can benefit from the results of our research. For example, edit operations that we discover in the context of edit operation mining can be used in knowledge-based recommender systems to repair models, to refactor models, or to merge models, or even for model completion [180]. Similarly—as for GitHub Copilot in the source code domain—model auto-completion can be used in several downstream activities, such as model generation and model explanation, for example, by adding a comment to a model, and triggering a generation of the comment content itself. Our approaches can therefore be directly applied in a modeling environment, or they can be applied to several of these downstream modeling activities. In this thesis, we will only cover

the concrete case of semantic lifting in more depth, and discuss other downstream activities only briefly.

1.6 Thesis Outline

This thesis is structured into several parts and chapters, each addressing different aspects of our research. Each part aims towards achieving the corresponding goal, formulated in Section 1.2.

The first part, **Model-driven Engineering and Software Evolution**, in Chapter 2, provides background about the field of Model-driven Engineering and Model-based Engineering, in particular, focusing on graph theoretical concepts and model transformations. In Chapter 3, an introduction to software and software model evolution is given. To develop a mathematical theory of software model evolution in Section 3.5, this part provides the required background including complexity measures in Section 3.2, graph mining in Section 3.3, and generative models and generative machine learning in Section 3.4. The mathematical theory developed in Section 3.5 serves as the theoretical foundation for software model evolution from a data-driven perspective, providing a common viewpoint for the research of this thesis.

The second part, **Case Study & Thesis Context & Datasets**, provides the setting and application context of our research. In Chapter 4, a real-world, industrial project for train control software at our industry partner will be introduced. We will study the challenges and possible root causes that are faced in this large-scale project. We will discuss the challenges of discovering features in a product family from a set of products, and we will point out limitations that this approach faces in a real-world industrial setting (see Section 4.4). The challenges arising from this case study were the initial motivation for the research presented in the following part. In Chapter 5, we will give an introduction to the datasets that will be used for the research presented in the following part. One of these datasets is directly derived from the case study presented in Chapter 4, connecting our empirical studies in the subsequent chapters back to our initial motivation.

Finally, in the third part, **Learning from Evolution for Evolution**, we study several use cases and propose approaches, in particular, edit operation mining using graph mining in Chapter 6, edit operation mining via generative models and pattern memorization in generative models in Chapter 7, and model (auto-)completion in Chapter 8. In particular, we will report on experiments to evaluate the proposed approaches, which can also be understood as experiments evaluating the proposed technologies—such as graph mining or large language models—for a particular use case—such as edit operation mining or model (auto-completion). This part could also be considered as a first empirical evaluation of the theory presented in Chapter 3, though more research is necessary to validate the theory, because the generalizability

and external validity of the experiments performed in the third part of the thesis is not appropriate for the general scope of the statements of the theory.

We will discuss concluding remarks and an outlook to future research directions in Chapter 9.

Part I

Model-driven Engineering and Software Evolution

2

Model-driven and Model-based Engineering

All models are wrong, but some are useful.

— George Box

Model-driven Engineering is an advanced software engineering approach that elevates models to the forefront of the development process [39, 186, 232]. This chapter provides a brief overview of Model-driven Engineering, as well as important concepts in Model-driven Engineering, such as model transformations, edit operations, and recommender systems in Model-driven Engineering. The chapter also formalizes concepts that are necessary for the understanding of the rest of this thesis.

2.1 The Fundamentals of Model-driven and Model-based Engineering

The first term that needs to be clarified when discussing Model-driven Engineering is the term *model*. Motivated by earlier definitions [39, 269], we define a model informally as a system that represents a simplified view of a *system under study*. We later provide a more formal definition of a model in Section 2.3.

Definition 2.1.1 Model (informal).
A model *is a system that represents an abstract (usually simplified) view of a system under study and allows to reason about the system under study without the need to consider it directly. A model can be* descriptive*, meaning that the system under study exists already, or* prescriptive*, meaning that the system under study does not exist (yet).*

Model-driven Engineering is often seen as a software development methodology [186, 232, 269], where (domain) models [104] serve as primary artifacts, guiding analysis,

simulation, and reasoning about the system under development, often culminating in the auto-generation of a portion of its implementation [186]. Model-driven Engineering adapts principles from traditional systems engineering to manage complexity and enhance productivity through model-centric automation and abstraction [232].

> **Definition 2.1.2** Model-driven Engineering – adapted from Kolovos [186].
> Model-driven Engineering *is the practice of raising models to first-class artifacts of the software engineering process, using such models to analyze, simulate, and reason about properties of the system under development, and eventually, often auto-generate (a part of) its implementation.*

Other definitions of Model-driven Engineering [39, 231, 232, 269, 307] include further characteristics such as model transformations, a strong focus on domain or abstraction, or round-trip engineering.

While Model-driven Engineering provides a comprehensive framework, it is often conflated with related but distinct concepts such as Model-driven Architecture (MDA), Model-based Engineering (MBE), and Model-driven Development (MDD).

- Model-based Engineering has a broader scope than Model-driven Engineering and describes a methodology where models play an important role in the engineering lifecycle of a (software) system, but are not necessarily the key artifacts and rather play a supporting role [39, 346].
- Model-driven Development, on the other hand, is a subset of Model-driven Engineering with a narrower focus on development activities like analysis, design, and implementation [39, 69].
- Model-driven Architecture, an initiative by the Object Management Group (OMG), is a specific incarnation of Model-driven Development that emphasizes the use of OMG standards and delineates three levels of model abstraction [38, 319].

Figure 2.1 depicts the relationship between these terms. In practice, the distinction between Model-driven Engineering and Model-based Engineering can be fluid, with the application context determining the degree to which model-centric methods are employed. For the purpose of this thesis, we adopt a pragmatic stance on Model-driven Engineering, recognizing its role as a guiding methodology while acknowledging the adaptability required to fit also different application domains as covered in this thesis. In fact, for the purpose of this thesis it is more suitable to adapt a more technical definition of Model-driven Engineering, also emphasizing the generalizability of the approaches proposed and discussed in part three of this thesis:

We will assume that models are typed over a given *meta-model*, defining the models abstract syntax and static semantics. On the one hand, this excludes engineering

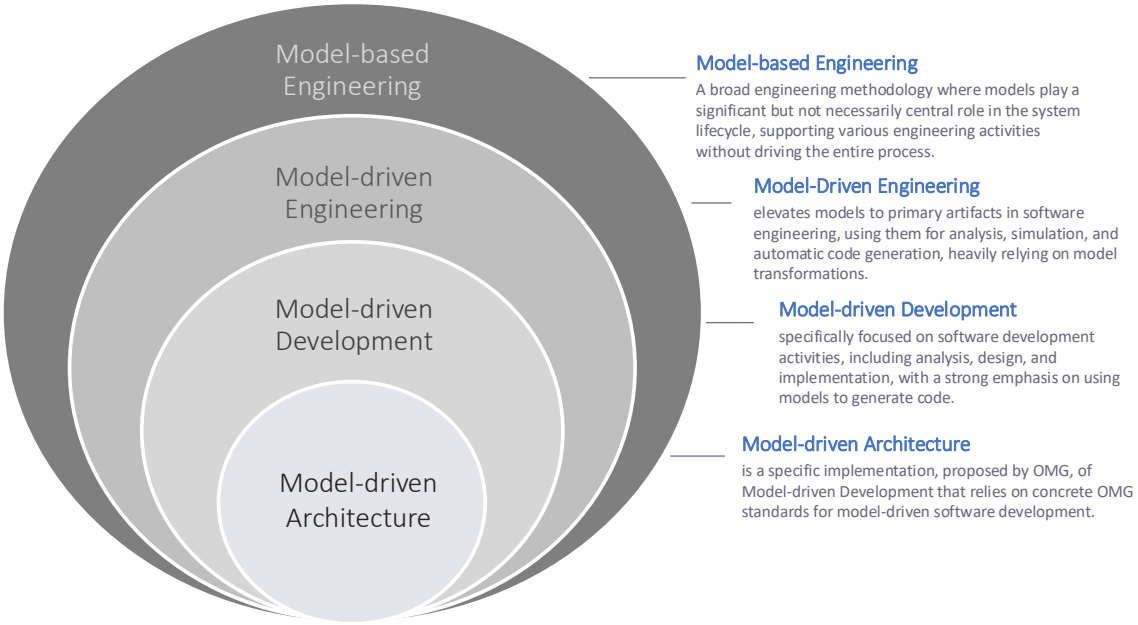

Figure 2.1: Onion model (according to [39]) for the terminology of Model-based Engineering (MBE), Model-driven Engineering (MDE), Model-driven Development (MDD), and Model-driven Architecture (MDA).

methodologies, where models are just used as a tool for documentation or communication, often as diagrams drawn with general purpose drawing applications such as Microsoft Visio, PowerPoint, or draw.io. On the other hand, this view includes methodologies outside of software engineering, where strictly typed models are used, such as in systems engineering, where models are used to describe the system architecture, mechanical and electrical computer-aided design, or in the natural sciences, where models are used to describe physical systems.

This leaves us with the following "working definition" for the thesis:

Definition 2.1.3 Model-driven Engineering "Working Definition".
Model-driven Engineering *is an engineering paradigm where* models *are the primary artifacts of the engineering process. Models are typed over a given* meta-model, *defining the models' abstract syntax and static semantics. Tooling is available to parse, validate, manipulate, and visualize a given model.*

We will also require that models are *versioned* and that the evolution of models is tracked, because the approaches presented in this thesis are based on the analysis of model histories. For a given model, we therefore have access to one or more previous versions of the same model.

2.2 Graph Theory

As we will see in Section 2.3, software models can be represented as *graphs*. In this section, we therefore introduce necessary concepts from graph theory. In this thesis, we will usually work with labeled graphs:

Definition 2.2.1 Labeled Graph.
A labeled directed graph G is a tuple (V, E, λ), where V is a finite set of nodes, E is a subset of $V \times V$, called the edge set, and $\lambda : V \cup E \to L$ is the labeling function, assigning labels from an alphabet L to nodes and edges. A graph is called directed, if the edge set E is consisting of ordered pairs of vertices, that is, (u, v) is the directed edge from u to v. A graph is called undirected, if an edge $(u, v) \in E$ implies the existence of the edge $(v, u) \in E$.

If we are only interested in the structure of a graph and typing is irrelevant, we will omit the labeling and only refer to the graph as $G = (V, E)$.

Similar to many other kinds of structures (e.g., sets), there is also a notion of containment for graphs, namely the definition of a *subgraph*:

Definition 2.2.2 Subgraph.
Given two graphs $G = (V, E, \lambda)$ and $G' = (V', E', \lambda')$, G' is called a subgraph *of G, written $G' \subseteq G$, if $V' \subseteq V$, $E' \subseteq E$, and $\lambda(x) = \lambda'(x)$ for each $x \in V' \cup E'$ (labelled over the same alphabet L).*

Being equal in the language of graphs is called *graph isomorphism*. Furthermore, a graph G_2 can be *(subgraph) isomorphic* to a subgraph of another graph G_1.

Definition 2.2.3 Graph Isomorphism.
Two graphs G_1 and G_2 are said to be isomorphic, *if there is a bijection $\phi : V_{G_1} \to V_{G_2}$ with*

$$(\phi(u), \phi(v)) \in E_{G_2} \Leftrightarrow (u, v) \in E_{G_1}. \tag{2.1}$$

A graph G_2 is said to be subgraph isomorphic to G_1, $G_2 \preccurlyeq G_1$, if G_1 has a subgraph that is isomorphic to G_2. The subgraph isomorphism ϕ is called induced, if furthermore equation 2.1 holds for each edge of G_1 with nodes in $\phi(V_{G_2})$.

A graph G can be decomposed into its so called (weakly) connected components, which are maximal subgraphs that can not be reached from each other by following edges.

Definition 2.2.4 Weakly Connected Component.
A (weakly) connected component *of a graph G is a subgraph $C = (V_C, E_C) \subseteq G$ of G in which every two vertices are connected by a path, that is, $\forall u, v \in V_C \colon \exists\, n \in \mathbb{N}$ s. t. $\{(v, v_1), (v_1, v_2), \ldots, (v_n, u)\} \subseteq E_C \cup \tilde{E}_C$, where $\tilde{E}_C$ is the set of all reversed edges, that is, $(u, v) \in E_C$ becomes $(v, u) \in \tilde{E}_C$, and every vertex in G that is connected to a vertex in V_C by a path is an element of V_C. For the sake of brevity we will use the term* component *for both—weakly connected components for directed graphs, or connected components for undirected graphs, if not stated otherwise.*

2.3 Modeling Concepts

In this section, we will review some basic concepts in Model-driven Engineering—central to this thesis—using the graph theory concepts introduced in the previous section. As usual in Model-driven Engineering, we assume that a *meta-model* specifies the abstract syntax and static semantics of a modeling language [307]. Additional constraints can be defined in the form of *well-formedness rules* or *invariants* [39]. Conceptually, we consider a model as a *typed graph* (also called abstract syntax graph), in which the types of nodes and edges are drawn from the meta-model. It is important to distinguish between the *concrete syntax*, which defines how the model is presented to a user by a tool, and the *abstract syntax* defined by the meta-model. A concrete model (i.e., an instance of the meta-model) can then be represented using the abstract syntax graph, or a concretely defined visual representation (e.g., a diagram or a textual representation).

Definition 2.3.1 Model, Meta-Model, Abstract Syntax, Concrete Syntax.
Given a label alphabet $L_{\mathcal{TM}}$, a meta-model *$\mathcal{TM}$ is a tuple $(V_{\mathcal{TM}}, E_{\mathcal{TM}}, \lambda_{\mathcal{TM}})$, where $V_{\mathcal{TM}}$ is a finite set of node types, $E_{\mathcal{TM}}$ is a finite set of edge types, and $\lambda_{\mathcal{TM}} : V_{\mathcal{TM}} \cup E_{\mathcal{TM}} \to L_{\mathcal{TM}}$ is the labeling function.*

A model *M conforming to $\mathcal{TM}$ is a tuple (V, E, λ), where V is a finite set of nodes, E is a subset of $V \times V$, and $\lambda \colon V \cup E \to L$ is the* labeling function*, which assigns a label to nodes and edges. M is* typed over *$\mathcal{TM}$ if there exists a type morphism (cf. Biermann et al. [37]) $\mu : V \cup E \to V_{\mathcal{TM}} \cup E_{\mathcal{TM}}$.*

The abstract syntax graph *of a model M is given by $G_M := (V, E, \lambda)$.*

A concrete syntax representation *of a model M is its visual* representation.

The abstract syntax is usually the basis for several tool functionalities, such as model editors, model checkers, or model transformations. A meta-model itself is often a model, typed over a *meta-meta-model*, which defines the abstract syntax of the meta-model.

A practical example of a meta-meta-model is the ECORE meta-model [310], which is used in the Eclipse Modeling Framework (EMF) [310] to define the abstract syntax of models in EMF. ECORE is an implementation of the *Meta-Object Facility* (MOF) [245], which is a standard by the Object Management Group (OMG) for defining meta-models.[1] One could then use ECORE to define a meta-model, for example, for (a subset of) UML. The models that follow this meta-model would then be valid UML models. The *concrete (visual) syntax* of a model is an external representation on the model that provides visual cues for the end-user of a model, typically via a tool. The concrete syntax is meant to allow for easier handling and understanding of the models, and therefore it may even hide information of the model. In the example of UML, the diagrams are based on the concrete syntax of the UML models.

Figure 2.2 illustrates how a simplified excerpt from an architectural model (cf. the case study in Section 6.1.4) in concrete syntax is represented in abstract syntax, typed over a given meta-model.

Remark

We abstain from a formal definition of typing using type graphs and type morphisms [37], though. Instead, to keep our basic definitions as simple as possible, we work with a variant of labeled graphs. In essence, we omit the complexity of type morphisms and type graphs, and assume for most part of the thesis that models are correctly typed. Furthermore, we will mostly work with the abstract syntax of models, setting aside tool and user-centric questions for the research questions of this thesis. We will discuss the implications of these simplifications per approach in part three of this thesis. More details on the concrete label representation can be found in the appendix in Section E.3.

1 More precisely, it has been proposed as standard for platform and language independent object exchange [245].

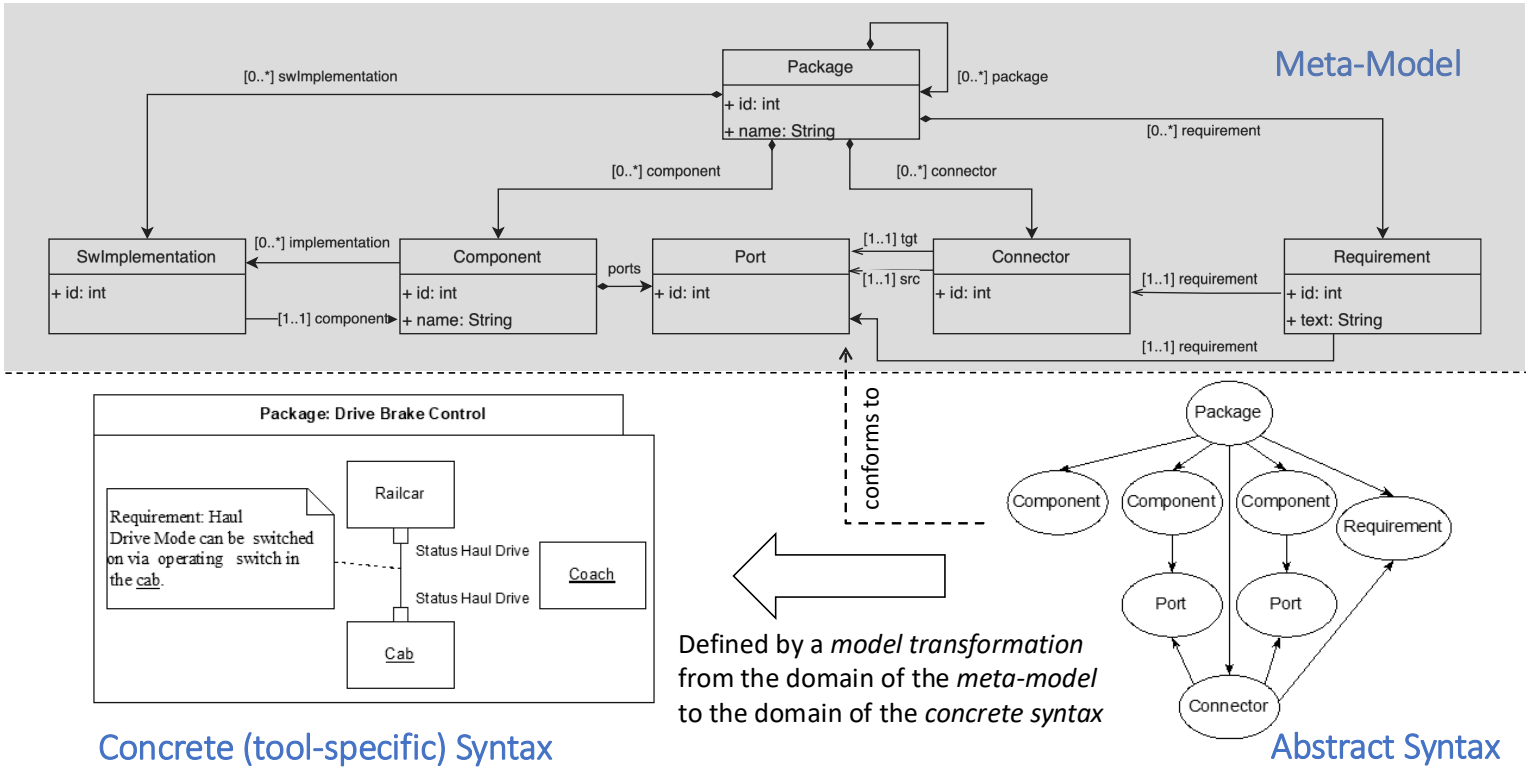

Figure 2.2: We consider models as labeled graphs, where labels represent types of nodes and edges defined by a meta-model. For the sake of brevity, the types of edges are omitted in the figure.

Given a concrete use case, the concrete label representation λ in Definition 2.3.1 can be defined to optimally support this use case. In the concrete example of edit operation mining, we focus only on type level, while in the example of model completion, we also encode attributes in the node labels. For example, a JSON-like representation of a node label could be used, which contains the type (i.e., the corresponding classifier in the meta-model) and all relevant attributes. Likewise, if only type information (e.g., the classifier name) is relevant for the use case, the node labels could correspond to the names of the classifiers from the meta-model. Depending on the use case, other label representations are conceivable. This flexibility in the label definition allows us, for the *evaluation* of a certain approach, to compare a predicted model to a ground truth model up to a certain *level of correctness*. That is, by omitting the label, we can compare two models structurally. Likewise, we can use only type information as label, to check for type correctness.

Another important concept in Model-driven Engineering is the notion of a *model transformation*. Since it is very central to this thesis, we will discuss it in more detail in Section 2.4.

2.4 Model Transformations and Edit Operations

In this section, we present an overview to model transformations—a key concept in Model-driven Engineering. Indeed, as we will see in this chapter, edit operations—the central concept for the evolution of software models—can be seen as a special kind of model transformation. We will formalize edit operation from a viewpoint that is relevant for this thesis, namely from a perspective that allows us to identify edit operations from model differences, that is, identifying edit operations after they have been applied to a model.

2.4.1 Introduction to Model Transformations

A *model transformation* is a function that maps a model conforming to a given meta-model to another model conforming to the same or a different meta-model. Model transformations play a crucial role in Model-driven Engineering for several reasons: they are used to transform models from one representation to another, for example, to generate code from a model, to refactor a model, or to migrate a model to a new version of the meta-model. Furthermore, as we will see later in this section, the evolution of models can be described as a sequence of model transformations, where each transformation represents a change in the model.

Definition 2.4.1 Model Transformation.
A model transformation *is a function* $t : \mathcal{M}_1 \rightarrow \mathcal{M}_2$*, where* $\mathcal{M}_1$ *and* $\mathcal{M}_2$ *are sets of all valid models (according to meta-models* $\mathcal{TM}_1$ *and* $\mathcal{TM}_2$*, respectively). A model transformation is typically defined by the meta-model of the source model and the meta-model of the target model, both following a common "meta-meta-model".*

In Model-driven Engineering, model transformations are usually defined by a *model transformation language* (e.g., QVT [246], ATL [153], or Henshin [19]), which is based on a *meta-meta-model* (e.g., the ECORE meta-model) that describes the syntax and semantics of the model transformation language, as well as the syntax and semantics of the models that are transformed. The advantage of this approach is that source domain and target domain are explicitly utilized within the model transformation definition, which highlights the domain languages in the model transformation and also allows for model transformations to be models themselves. This is different from general purpose (programming) languages, where the domain is often implicit in the code. This approach is in line with the Model-driven Engineering philosophy of raising models to first-class artifacts of the software engineering process.

Figure 2.3 shows an illustration of the concept of model transformations (based on Brambilla et al. [39]).

Model transformations can be classified along several dimensions of which we want to highlight the following three [39, 224]:

Endogenous vs. Exogenous An *endogenous model transformation* transforms a model conforming to a meta-model to another model conforming to the same meta-model. An *exogenous model transformation* transforms a model conforming to a meta-model to another model conforming to a different meta-model.

Vertical vs. Horizontal A *vertical model transformation* transforms a model from a higher level of abstraction to a lower level of abstraction or vice versa.

Syntactic vs. Semantic A *syntactic model transformation* transforms a model without considering the meaning of the model elements. A *semantic model transformation* transforms a model while considering the meaning of the model elements.

Another dimension often considered [39] is whether a new model is generated by the model transformation (also called *out-place* model transformation) or whether the model is rewritten (e.g., via additions and deletions) (also called *in-place* model transformation). Focusing on the first two dimensions (i.e., endogenous vs. exogenous and vertical vs. horizontal), in Table 2.1 we give an example for each combination of the endogenous vs. exogenous and vertical vs. horizontal dimensions [224].

The example of endogenous, horizontal model transformations we will consider in depth in what follows: In this thesis, we will focus on the evolution of models, and we will use the term *edit operation* to describe a modification to a model that results in another valid model. This term highlights that typically users (human

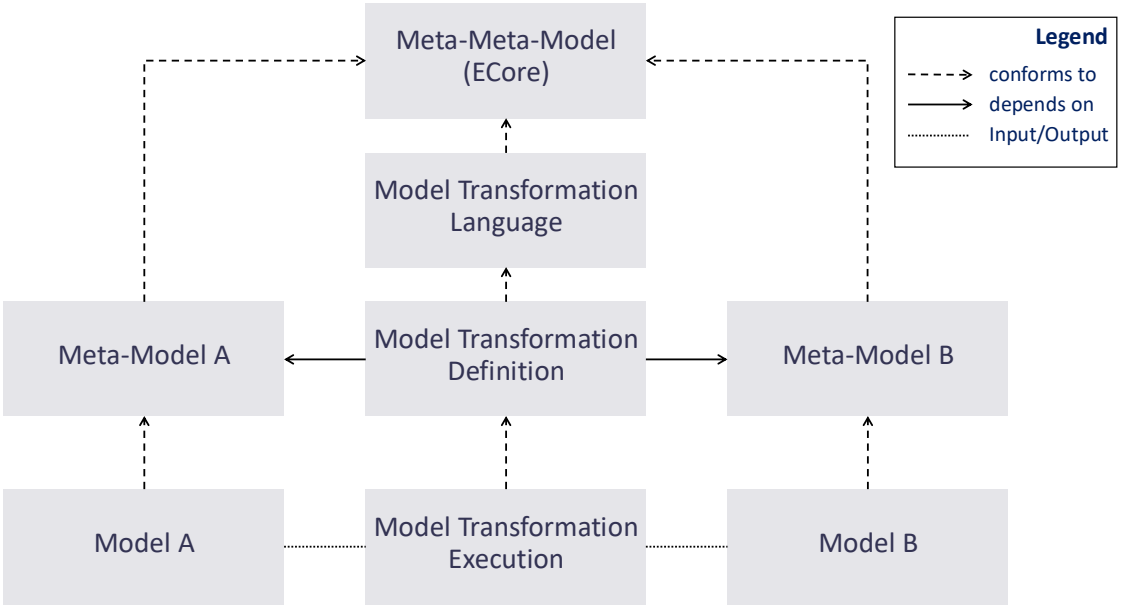

Figure 2.3: This graphic (based on a book by Brambilla et al. [39]) visualizes the definition of model transformations in Model-driven Engineering. Model transformations transform a source model (Model A) to a target model (Model B). A model transformation is defined by the meta-model of the source model (Meta-Model A) and the meta-model of the target model (Meta-Model B). In Model-driven Engineering both, source- and target meta-model are described by a unified "meta-meta-model". This meta-meta-model also serves as the basis for the definition of model transformations via a model transformation language.

users or intelligent assistants) perform *(tool) operations* to edit a model. For example, a human user of a UML modeling tool might add a new class to a class diagram, or a developer might refactor a class diagram by splitting a class into two classes. In general, edit operations can be informally understood as editing commands that can be applied to modify a given model. Edit operations can be described as endogenous model transformations, because source and target meta-model are the same. They can be described as horizontal model transformations, because an edit operation does not change the level of abstraction.

Definition 2.4.2 Edit operation (informal).
An edit operation is an endogenous, horizontal model transformation offered by a tool to a user. We also consider the composition of two edit operations (i.e., executing one edit operation after another edit operation) as edit operation.

In theory, other *agents*, for example, an intelligent modeling assistant (see Section 2.5) could also execute edit operations, provided by the modeling tool.

Table 2.1: Example for the classification of model transformations along the dimensions of endogenous vs. exogenous and vertical vs. horizontal (adapted from Mens et al. [224]).

	Vertical	Horizontal
Endogenous	Model Refinement	Edit Operations
Exogenous	Java Code from Class Diagram	ERD to Class Diagram

2.4.2 Formalization

This section shares material with Tinnes et al. [326].

Edit operations have been formalized by Ehrig et al. [90] and Biermann et al. [37] using typed attributed graphs. The focus of their formalization is to mathematically precisely define the application of edit operations to models. In this section, we provide a new, orthogonal formalization that is better suited for studying the approaches in this thesis.

In this section, we will provide a new formalization of edit operations focusing on simplified graph representations of models. We can leverage these graph representations more easily in the study of model evolution in this thesis, for example, in the context of edit operation mining.

Earlier formalization of endogenous model transformations (and therefore also edit operations) are based on typed attributed graphs [37, 90]. The basic idea of typed graphs is to define a graph homomorphism (i.e., a function from a typed graph G_M to a type graph TG (defined by the meta-model $\mathcal{TM}$)) satisfying some structure preserving constraints. Details of this formalization are given in the work by Ehrig et al. and Biermann et al. [37, 90]. In the appendix in Section B, we will also provide more details on their formalization.

In this work, we sidestep questions regarding preservation and checking of model validity. We assume that the modeling tool already takes care of checking the correct typing of the models, and we therefore expect that the models are correctly typed. We therefore work with a simplified graph representation of the models in which the abstract syntax graph is just a *labeled directed graph* (cf. Definition 2.3.1). As described earlier, in Model-driven Engineering, the language for a software model (i.e., its abstract syntax and static semantics) is typically defined by a meta-model $\mathcal{TM}$. We denote by $\mathcal{M}$ the set of all valid models (according to some meta-model). We refer to the set of all directed labeled graphs by $\mathcal{G}$.

Two valid models with the same meta-model can be compared, that is, it is guaranteed that a *model difference* can be computed between them. This difference can be computed based on their abstract syntax graph.

Definition 2.4.3 Structural Model Difference.
A structural model difference Δ_{mn} *of a pair of model versions m and n is obtained by matching corresponding model elements in the model graphs G_m and G_n (using a model matcher [313], for example, EMFCompare [43] or SiDiff [284]). The matching results in added elements (the ones present in G_n but not in G_m), removed elements (the ones present in G_m but not in G_n), and preserved elements that are present in G_m and G_n. Furthermore, attributes can change their values. In these cases, we add auxiliary nodes to the modified nodes and edges to represent the attribute changes with their value in m and n.*

In order to be uniquely defined, this definition assumes a deterministic model matcher, that is, given two models $m, n \in \mathcal{M}$, we get a unique structural model difference Δ_{mn}.

The structural model difference can be represented as a *difference graph* [244] $G_{\Delta mn}$, where the nodes carry some additional information `Add`, `Preserve`, or `Remove`, and matching elements (i.e., the preserved ones) from G_m and G_n are unified with each other (i.e., they will be present only once).

In practice, we add this information to the labels of the original abstract syntax graphs. For example, one could have an additional JSON attribute `changeType` in the node and edge labels, which can be set to `Remove`, `Preserve`, or `Add`. Attribute changes will have a `changeType` `Change` and references to the original and the new value via `valueBefore` and `valueAfter` information in the label of the auxiliary node. Note that this is a design choice in our implementation and other realizations of a difference graphs and changed attribute values, in particular, would be conceivable.

Note that the change graph comprises all information of the left hand side model m and the right hand side model n. In particular, it contains all preserved nodes and edges. Especially for large models and rather local changes the change graph can be very large compared to the actual changes in the model. This motivates our definition of a *simple change graph* as the smallest subgraph comprising all changes in the difference graph $G_{\Delta_{mn}}$.

Definition 2.4.4 Simple Change Graph.
Given a difference graph $G_{\Delta_{mn}}$, a simple change graph $SCG_{\Delta_{mn}} \subseteq G_{\Delta_{mn}}$ *is derived from $G_{\Delta_{mn}}$ by first selecting all the elements in $G_{\Delta_{mn}}$ representing a change (i.e., added, removed nodes and edges) and, second, adding preserved nodes that are adjacent to a changed edge. The simple change graph is the smallest subgraph of $G_{\Delta_{mn}}$ containing all changed nodes and edges.*

Figure 2.4 shows an example of a simple change graph derived from a difference graph.

In principle, the simple change graph can be understood as a template for a model transformation: First, the preserved and removed nodes are matched in the source

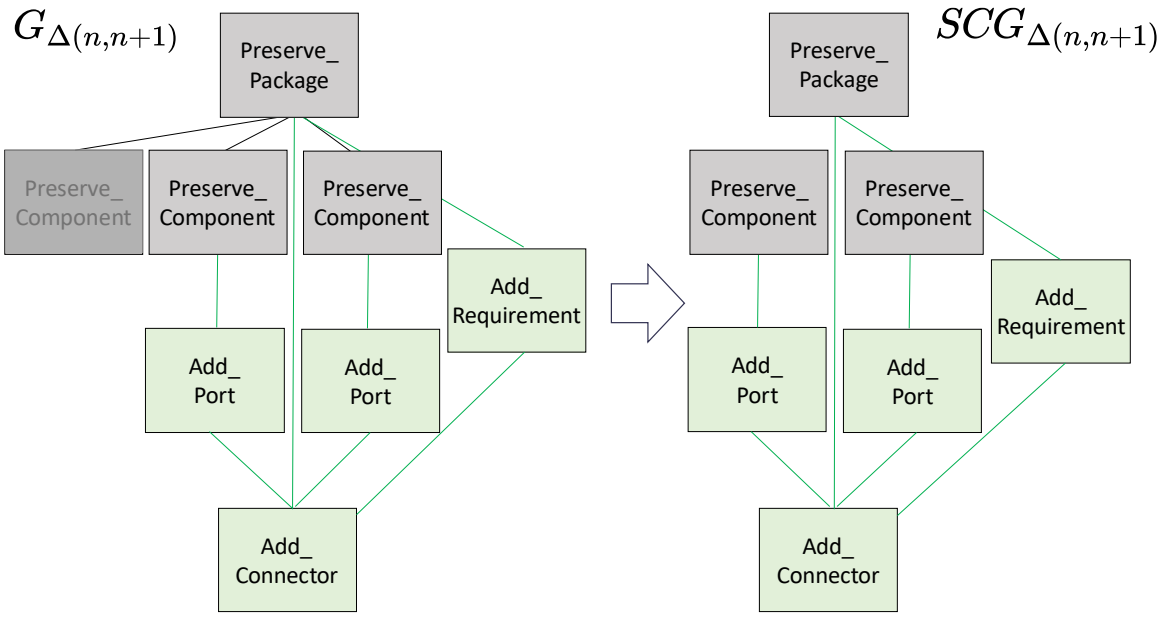

Figure 2.4: A simple change graph is derived from the difference graph, by removing all nodes, and edges that are not directly connected to other changed edges or nodes.

model, then the removed nodes are removed, and the added nodes are added to the source model gluing them to the preserved nodes as defined by the simple change graph. We will make this construction more precise in what follows. A full formalization of model transformations is given in the appendix in Section B.

We define a *model transformation application* simply as a pair of models, where the first model is the source model and the second model is the target model of the transformation.

Definition 2.4.5 Endogenous model transformation application.
An endogenous model transformation application $t = (m, n) \in \mathcal{M} \times \mathcal{M}$ *is a pair of models. We call* m *the* source model *and* n *the* target model *of the transformation and* $\mathcal{T} \stackrel{def}{=} \mathcal{M} \times \mathcal{M}$ *the space of endogenous model transformation applications.*

This definition only describes a concrete application of a model transformation but does not provide the definition in terms of the common meta-model $\mathcal{TM}$. In practice, we are interested in a more reusable template-like definition of a model transformation.

To this end, we define a function $SCG\colon\ \mathcal{T} \to \mathcal{G}$ that takes a model transformation application t (i.e., a pair of models) as input and returns the simple change graph for the corresponding model difference.

We can use this map SCG to define an equivalence relation on $\mathcal{T}$ by

$$\begin{aligned} t_1 = (m, n) &\sim t_2 = (k, l)\,, \qquad \text{if and only if} \\ SCG_{\Delta_{mm}} &= SCG_{\Delta_{kl}}. \end{aligned}$$

An equivalence relation is a reflexive, symmetric, and transitive relation, that is,

- $t \sim t$ for all $t \in \mathcal{T}$ (reflexivity),
- $t_1 \sim t_2$ implies $t_2 \sim t_1$ for all $t_1, t_2 \in \mathcal{T}$ (symmetry),
- $t_1 \sim t_2$ and $t_2 \sim t_3$ implies $t_1 \sim t_3$ for all $t_1, t_2, t_3 \in \mathcal{T}$ (transitivity).

Indeed, any relation defined via a map (such as SCG) is an equivalence relation, because the map is deterministic and therefore the relation is reflexive, symmetric, and transitive.

Model transformation applications that are equivalent under this relation are said to be in the same *equivalence class*. The set of all model transformation applications can then be partitioned into equivalence classes, where each equivalence class contains all model transformation applications that have the same simple change graph. For equivalence relations, we can define the *quotient set* $\mathcal{T}/\sim$, which is the set of equivalence classes. This quotient set $\mathcal{T}/\sim$—by construction—is set isomorphic to the set of simple change graphs, that is, the range of the map SCG.

We can use this construction to formally define the concept of an endogenous model transformation.

Definition 2.4.6 Endogenous model transformation.
An endogenous model transformation (definition) *is an equivalence class in the set* $\mathcal{E} \stackrel{def}{=} \mathcal{T}/\sim$. *The endogenous model transformation is therefore a set of model transformation applications that have the same simple change graph.*

We can also interpret an endogenous model transformation as a template for a rule to transform a model m into a model n: For a simple change graph, we call the subgraphs of "Remove" and "Preserve" nodes the left-hand side graph L, and the "Add" and "Preserve" nodes the right-hand side graph R. The subgraph of preserved nodes and edges K is then embedded in L and R, respectively. We denote this embedding by $L \hookleftarrow K$ and $K \hookrightarrow R$ ($\hookleftarrow$ denotes an injective homomorphism, i.e., an embedding). The embedding of L and R along the preserved nodes K (i.e., $L \hookleftarrow K \hookrightarrow R$) in the simple change graph defines how to remove the "Remove" nodes from m and glue the "Add" nodes along K. Given an endogenous model transformation ε and a concrete model m, one can define a matching match: $L \hookrightarrow G_m$, and perform the removal of "Remove" nodes and the gluing of "Add" nodes as defined by the simple change graph corresponding to ε, and then set concrete attributes. This yields the corresponding model n with $(m, n) \in \varepsilon$, and this way an endogenous model

transformation $\varepsilon \in \mathcal{E}$ can be interpreted as a template for a model transformation application in agreement with previous constructions [37, 159, 324]. We therefore also write $m \xrightarrow{\varepsilon} n$ to denote a concrete endogenous model transformation in the equivalence class $\varepsilon \in \mathcal{E}$.

Originally, graph transformations have been defined more generally [89] allowing for a non necessarily injective graph morphsim from K to the left and right-hand side graphs L and R (see Section B for details). Still, relying on injective graph morphisms simplifies the construction of the model transformation and still, the entire space of valid models can be reached by a sequence of endogenous model transformations with the definitions above.

Remark 2.4.1
The graph labeling function λ allows us to define the scope of the model transformation. For example, if we are only interested in the type of the nodes and edges, we can omit the attributes from the label. Likewise, if we are interested in the attributes, or only want to set them during execution time, we could define placeholders for the attribute values in the labels. Therefore, the procedure above defines a model transformation language only up to the concrete label representation and leaves some freedom for templating. For details on the graph labeling functions used in our experiments, see Section E.3 in the appendix.

The set of model transformations obtained by this construction is huge—infinite to be more precise—and it contains also transformations such as constructing a large model from scratch (i.e., taking $m := \bot$, the empty model and $n \in \mathcal{M}$ a large model). In practice, only a small subset of these transformations for a given meta-model can be observed. For example, not every model that is valid according to a meta-model is actually useful in practice and therefore also not the model transformations that construct these models from scratch. In this thesis, we are interested in a subset of the set of all endogenous model transformations, that is, the ones which can actually be observed and are *meaningful*. These are the ones that are actually applied by a user or an intelligent assistant—or compositions thereof. We will refer to this subset as the set of *empirical edit operations* or, for short, edit operations, because they are the result of observable edits to a model.

A transition from $m \xrightarrow{\varepsilon} n$ can usually also be obtained by a sequence (also called *edit script* [164])

$$m \xrightarrow{\varepsilon_1} m_1 \xrightarrow{\varepsilon_2} \dots \xrightarrow{\varepsilon_{k-1}} m_{k-1} \xrightarrow{\varepsilon_k} n.$$

Definition 2.4.7 Generator.
A set $S \subset \mathcal{E}$ is called a generator *for $\mathcal{E}$, if every model can be reached by a sequence of edit operations in S, that is,*

$$\forall\, m \in \mathcal{M}.\ \exists\, \varepsilon_1, \ldots, \varepsilon_{k_m} \in S,\ m_1, \ldots, m_{k_m-1} \in \mathcal{M}$$
$$\textit{such that} \quad \bot \xrightarrow{\varepsilon_1} m_1 \xrightarrow{\varepsilon_2} \ldots \xrightarrow{\varepsilon_{k_m-1}} m_{k_m-1} \xrightarrow{\varepsilon_k} m.$$

An example for a generator S is the set of *elementary edit operations* that can be derived from a meta-model, as given by Kehrer et al. [166]. This set is finite and contains rather fine-grained edit operations.

Theorem 2.4.1
Any subset $E \subset \mathcal{E}$ can be completed to a generator by joining it with an existing generator S (e.g., the set of elementary edit operations).

This is obvious from the definition of the generator, because we can always take $\varepsilon_1, \ldots, \varepsilon_{k_m} \in S \subset S \cup E$ in the definition of the generator.

This theorem together with earlier results that elementary edit operations can be derived from the meta-model [166] ensures that a set of edit operations can always be completed to a generator.

In this work, we explore the continuum between elementary edit operations and the set $\mathcal{E}$ of all model transformations. One of our goals is to identify those higher-level edit operations whose effect is actually observable in model histories, providing empirical evidence that these are meaningful edit operations from a modeler's point of view. To give a concrete example for the difference between $\mathcal{E}$, a generator S, and edit patterns, consider the modeling language SysML, which is used in system engineering [248]. The set $\mathcal{E}$ would then include very fine-grained elementary edit operations from S, for example, adding a `port` to a `component` or adding a `connector` between two `ports`, but also very large edit operations, for example for setting up the entire system architecture for a train control software on the scratch. An empirical edit operation could be given, for example, by "adding an interface", that is, adding source port, target port, and connector in one step.

2.5 Intelligent Modeling Assistants and Recommender Systems in Model-driven Engineering

Central to this thesis is the study of Intelligent Modeling Assistants in the field of software model evolution. In this section, we will introduce the concept of *Intelligent Modeling Assistants* and their relation to *recommender systems* in the context of Model-driven Engineering.

Savary-Leblanc et al., in their mapping study on recommender systems in software engineering [280], define:

> **Definition 2.5.1** Intelligent Software Assistant.
> Bots *are systems featuring at least one of the following characteristics:*
>
> - *automates one or more feature(s) or capabilities of a system,*
> - *performs one or more function(s) that a human may do,*
> - *interacts with a human or other agents.*
>
> Software bots *are bots for software engineering. An* Intelligent Software Assistant *is a software bot that provides users with valuable knowledge to help them identify, understand, or solve a software engineering problem.*

We are not aware of any formal definition of an Intelligent Modeling Assistants. For this work, we will consider Intelligent Modeling Assistants as a subclass of intelligent software assistants that are specialized in the domain of software modeling.

> **Definition 2.5.2** Intelligent Modeling Assistant.
> *An* Intelligent Modeling Assistant *is a software bot that provides users with valuable knowledge to help them identify, understand, or solve a problem in the domain of software modeling.*

Mussbacher et al. [235] propose a reference framework for Intelligent Modeling Assistants. In their framework a sociotechnical modeling system (basically the modeling tool) is coupled via a communication infrastructure with the Intelligent Modeling Assistant. This communication infrastructure also synchronizes context information between the sociotechnical modeling system and the Intelligent Modeling Assistant, which maintains a context shadow (i.e., a representation of the context from external system and relevant model context). The Intelligent Modeling Assistant has data acquisition and data production interfaces that connect it to data, information, knowledge[2], as well as human experts.

Two years before the publication of the reference framework for intelligent modeling assistants, in 2018, Cabot et al. [50] proposed the vision of *cognifying* Model-driven Engineering, arguing that cognification has the potential to turn the return on invest for many use cases from negative to positive.

2 For definition and relationship of data, information, and knowledge, in this context, see the DIKW pyramid: https://en.wikipedia.org/wiki/DIKW_pyramid (last accessed: September 23rd, 2024)

Definition 2.5.3 Cognification – adapted from Cabot et al. [50].
Cognification *is the application of knowledge to boost the performance and impact of a process.*

The authors stress that cognification is not limited to artificial intelligence. Also, integrating human knowledge (e.g., via crowdsourcing) is an example for cognification. Intelligent Modeling Assistants can be seen as an example for cognification in Model-driven Engineering, although cognification might even be a broader vision.

Next, we also have to clarify the relation of Intelligent Modeling Assistants to recommender systems.

Definition 2.5.4 Recommender Systems.
A recommender system *is an information filtering system that provides a* user *with a subset of information that is relevant to the user, given the current context.*

Compared to recommender systems, software bots are a rather new concept. Similar to Intelligent Modeling Assistants, recommender system support a user in a specific task. One of the main differences is the focus of software assistants on the concept of *knowledge* instead of *information*. This stressed the subjective nature of the information provided by the software assistant. Furthermore, recommender system historically have been collaborative filtering, content-based, or knowledge-based approaches [280]. Model completion, domain chatbots, model generation, or other rather recent developments, even though they fit into above recommender system definition[3], lead to a new terminology including terms such as "bot", "assistant", "copilot", highlighting the active role of the system in the interaction with the user.

Remark 2.5.1
In regard to the terminology of data, information, *and* knowledge, *it is our conviction that data (i.e., the raw material) already contain information. This will be demonstrated in detail in Section 3.2.2. In other words, a description of the data can frequently be derived from the data themselves. It is, of course, impossible to extract "all information"—indeed, such a notion of "all information" is not useful [303]. For example, some description (e.g., properties of the intent), of the data will only become available at a later point in time. The type of information that can be extracted from data is contingent upon the context, which may be represented as data, information, or knowledge. Knowledge we see as a subjective or contextualized form of information. As with the relationship between model and meta-model, having a meta-model of the*

3 For example, transformer-based language models [330] like ChatGPT can be seen as a recommender system, because they provide a subset of information (e.g., all valid word combinations) that is relevant to the user, given the current context.

meta-model allows for an integration of previously "incompatible" information. That is, having another layer above information is necessary to talk about information on a meta-level. In light of the challenges in defining the boundaries between information and knowledge and the limited value this thesis would gain from doing so, we refrain from using the term "knowledge". Instead, we reserve it for its conventional usage, such as in the context of knowledge-based systems, including those exemplified by semantic web technologies. In contrast, the distinction between data and information can be articulated with considerable precision, and this will be demonstrated in the forthcoming Section 3.2.2. We consider information as data described by a model[a]*—which itself is some form of data. When we use the term* data-driven *we refer to the process of transforming data into—at least—information.*

a Unfortunately, depending on the research field, there are many alternative terms for *model* or different concepts overlapping in their meaning, for example, *hypothesis*, *description*, *schema*, *context*, etc. Also note that the term *model* as used in Model-driven Engineering has a very similar meaning in the sense that it is a description of the (observation of the) realized system (i.e., data).

In this thesis, we focus on data-driven recommender systems in the context of Model-driven Engineering, though we acknowledge that the broader vision of cognification [50] and Intelligent Modeling Assistants, as given in the reference framework by Mussbacher et al. [235] is a valuable vision the community should strive towards and align on. It has to be noted that a classical recommender system also fits into the reference framework, with less focus on the interactive nature of assistants. As outlined in the introduction in Chapter 1, we focus on the evaluation of the merits of data-driven technologies for several tasks in Model-driven Engineering. Consequently, we have to side-step user interaction questions in this thesis. Indeed, this is also one of the merits of such a reference framework as it allows the decomposition of the vision of an Intelligent Modeling Assistant into several aspects that can be addressed individually with a clear focus and clearly locate a certain research in the broader vision.

3

Software and Software Model Evolution

We are going to die, and that makes us the lucky ones. Most people are never going to die because they are never going to be born...

— Richard Dawkins

Evolution in Software Engineering has been largely studied from an engineering perspective in the past. That is, most of the questions discussed are rather of the form "How to engineer a system for certain qualities and longevity?" and less a science perspective of "How does software evolve?" or even stronger "How to predict the evolution of software?". Although most of the content provided in this chapter is not novel per se, the viewpoint and the relationships between information theory and software evolution, as established in this chapter, have not been discussed in the literature before. To the best of our knowledge, no attempt of unifying theories of software model evolution, generative models, and pattern mining has been made yet—we are even not aware of any similar theory in the domain of source code or software in general.

3.1 Introduction to Software Evolution

This thesis is concerned with the evolution of software systems and software models. The present section will establish common understanding of software evolution and provide necessary background and examples. Software systems embedded in the real world evolve [201]. As part of a sociotechnical ecosystem, software systems must adapt to their environment and meet the needs of their stakeholders. For a single software system, evolution affects several of its constituents:

- *Software Languages*: Software languages are subject to change, for example, because of emerging paradigms, shortcomings of the language, or new requirements from the language users.

- *Software Libraries*: Software libraries are subject to change, for example, because a new technology is emerging, or a certain domain is adapting.
- *Software Runtime*: Software runtimes are subject to change, for example, because of operating system changes, security patches, changes to the physical setup of a system, changes of the software language, et cetera.
- *Software*: A single software system is subject to change, because of new requirements, bug fixes, or changes in the environment.

Furthermore, software development processes, and consequently software development environments, tools, and everything used to perform tasks in the development lifecycle, are subject to change.

Additionally, it is important to distinguish between the evolution of a single system[1] and that of an entire ecosystem. It might be more appropriate to refer to the change process of a single system as *aging* [251] and that of an ecosystem as *evolution* [200, 201, 221]. Nevertheless, today, the term software evolution is used for the evolution of a single software system as well [221] and some authors emphasize the fact that this evolution is the inevitable consequence of the embedding of the software system into the real world [200, 201, 222].

In 1974, 1980, and 1996 Lehman formulated eight laws of software evolution [200, 201] that describe the evolution of software systems:

I Continuing Change: E-type systems must be continually adapted else they become progressively less satisfactory.

II Increasing Complexity: As an E-type system evolves its complexity increases unless work is done to maintain or reduce it.

III Self Regulation: E-type system evolution process is self-regulating with distribution of product and process measures close to normal.

IV Conservation of Organizational Stability (invariant work rate): The average effective global activity rate in an evolving E-type system is invariant over product lifetime.

V Conservation of Familiarity: As an E-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its

1 It is beyond the scope of this thesis to discuss aspects of systems theory. Anyway, it is important to note that a system can be understood as a set of interacting or interdependent components forming an integrated whole. Many of the software systems built nowadays are open systems and therefore interact with their environment, that is, they are *embedded* in a larger, surrounding system. Lehman [200, 201] called the corresponding programs *E-type* programs. When we talk about a single software system, we typically refer to an E-type system, with its own development organization that defines the boundary of this software system. The boundary is usually defined by a set of requirements.

content and behavior to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence, the average incremental growth remains invariant as the system evolves.

VI Continuing Growth: The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime.

VII Declining Quality: The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.

VIII Feedback System: E-type evolution processes constitute multi-level, multi-loop, multiagent feedback systems and must be treated as such to achieve significant improvement over any reasonable basis.

The evolution of a single software system and the evolution of the surrounding ecosystem are tightly coupled: To be successful, a single software system has to (self-)adapt to its environment,[2] and the environment is shaped by the (software) systems that are part of it. This, again, leads to change of the ecosystem and ultimately to a change of the single software system.[3] This observation is reflected in Lehman's first law of software evolution [200, 201]: "A program that is used and that, as an implementation of its specification, reflects some other reality, undergoes continual change or becomes progressively less useful. The change or decay process continues until it is judged more cost-effective to replace the system with a recreated version."

Besides this adaption to the environment, a single software system evolves on its own,[4] because it has to converge to its desired state—defined by the requirements and desired system qualities. Furthermore, the software system is undergoing some groomative changes, to ensure the system is maintainable[5] and extendable [60]. According to Lehman's second law of evolution [200] the complexity of a software system embedded in the real world is inevitably increasing unless effort is spent to reduce complexity.

What typically happens in the lifetime of a single software system is, that, according to Lehman's first law, a software system is undergoing inevitable change. This

2 See Lehman's 8th law of software evolution.

3 From a system theory perspective, a software system is a feedback system that tries to optimize its outputs based on feedback it receives from its environment (i.e., the larger system it is embedded in). Therefore, the evolution of the surrounding system has an impact on the evolution of the constituent system via this feedback. A system adapts to its environment, because the environment is changing. Self-organization and feedback is a typical emergent behavior for many complex systems.

4 This self-organizing behavior of an *E-type* system is covered by Lehman's 8th law. In systems theory, one could think of free energy that is utilized to reduce the entropy of the system, that is, increase the order of the system.

5 Some authors [60] distinguish between evolution and maintenance. For example, some define software evolution as changes visible to the user. We do not make this distinction here, although we acknowledge the usefulness of a more detailed taxonomy for several discussions.

change, according to Lehman's second law leads to increasing complexity of the software system. Lehman's sixth law of continuing growth, with the assumption that its basically impossible to add functionality without increasing the complexity, means that the complexity of the system will increase, even if work is done to reduce the complexity. One might be tempted to assume that one has to proportionally increase the workforce to be able to maintain the software system. Unfortunately, at some point, the law of diminishing returns comes into play, ultimately leading to a point where, from an economic point of view the system becomes unprofitable. In case of (too) high complexity, this is also covered by Brooks's law [40]: "Adding manpower to a late software project makes it later." Anyway, also by Lehman's fourth law, we see that (average) work force in the project is invariant over time. When a system becomes unprofitable, the system will be either phased-out, or, if the system is still critical for other reasons—for example, because it is required to sell another profitable product or the system responsibility is still required, and no replacement is available—it becomes a *legacy software system*.

For some systems (e.g., database systems or operating systems) the growth by new requirements can be handled, and the systems become rather stable and long-living. In general, as long as the system's architecture is still suitable to incorporate new requirements without increasing the entropy of the system too much, growth can be handled. On the other hand, if there is an architectural mismatch [109] to use existing (sub-)systems to realize new requirements either the entropy increases or efforts are necessary to adapt the software system's architecture. In other words, as long as a system is still resilient to changes, it can be maintained and extended.

The evolution of an entire ecosystem typically entails the evolution of its constituents, the emergence of new systems, and the decline and disappearance[6] of old systems. Even for a declining system some aspects or concepts of it might "survive"—or might have been adopted—in new systems. That is, software systems consist of interacting or interdependent components that realize *patterns* or *concepts* that are not unique to a single system but are shared among several systems.[7]

Example
Let us consider the concrete case of banking systems, developed in the 1960s until the present day, highlighting the interplay between technological advancement, regulatory requirements, and software engineering paradigms (see Figure 3.1).
The development of banking information systems started even before the standardization of programming languages and software engineering methodologies. In the end of the 1960s and beginning of 1970s COBOL became more prominent as a programming language. Still, the systems major purpose has

6 This disappearance of a previously established system is sometime referred to as *creative destruction*.
7 This idea is also one of the key principles in the study of software architecture [109, 143, 146, 187, 189].

been internal transaction processing and reporting. These systems faced significant challenges as the financial landscape evolved. Stringent anti-money laundering (AML) regulations and fraud detection requirements necessitated sophisticated transaction monitoring capabilities. Concurrently, the rise of internet technologies and enterprise resource planning systems demanded robust integration mechanisms, including web interfaces, APIs, and mobile applications. The shift towards object-oriented programming in the 1990s and early 2000s saw languages like Java and C++ supplanting COBOL. This transition, coupled with the adoption of web-based software design principles, significantly influenced application communication technologies.
As COBOL expertise became scarce, maintaining these systems grew increasingly complex and costly. Despite their critical role in core business operations, these systems gradually became "legacy systems"—essential yet expensive to maintain due to outdated architecture. Organizations faced the dilemma of balancing short-term disruption of system replacement against long-term benefits of reduced maintenance costs and improved technological alignment.

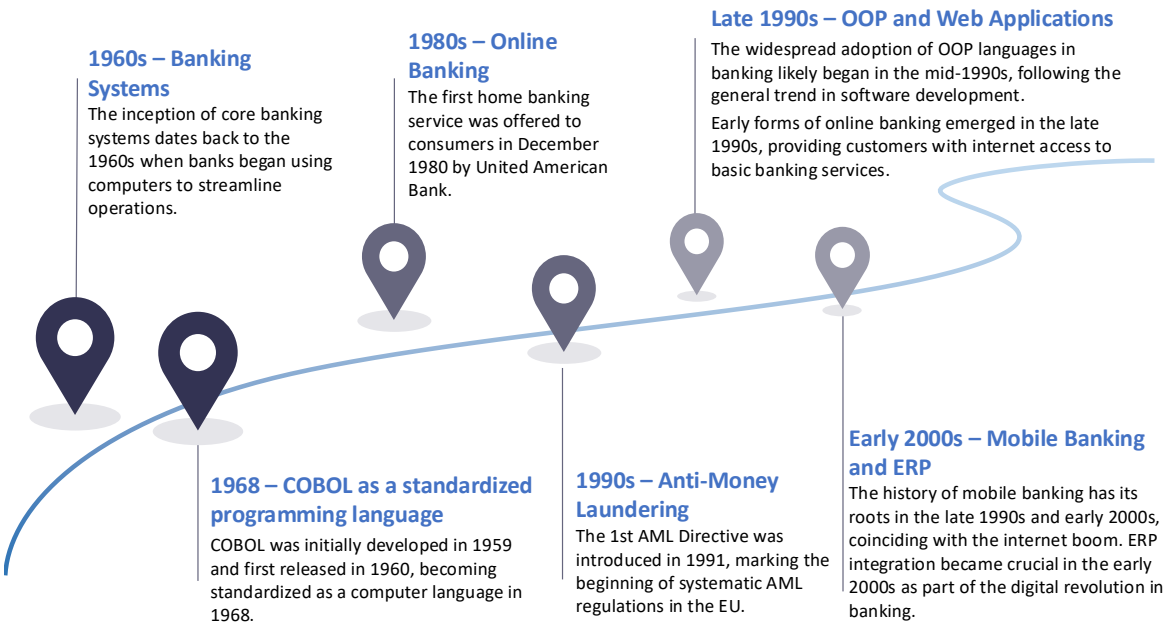

Figure 3.1: A timeline depicting events that are relevant for the evolution of banking information systems.

In contrast, as the following example demonstrates, other software systems developed around the same time, have faced less change.

Example
Database management systems like Oracle Database, or IBM DB2, developed in the late 1970s, have demonstrated remarkable adaptability over the decades. Unlike banking systems, database management system evolved more gradually via incremental improvements—such as queuing functionality. Key advancements included the transition from hierarchical to relational models, the incorporation of object-oriented features, and more recently, the adaptation to handle big data and cloud environments. Despite these significant changes, database management systems maintained stable core interfaces, allowing organizations to upgrade without major disruptions.

Contrasting these two examples highlights the challenges of maintaining long-lived software systems in dynamic technological and regulatory environments, and demonstrates the challenges of forward-thinking and designing a system for change.

Definition 3.1.1 Software Evolution.
Software Evolution *is the process of change of a software system over time. This process is driven by internal and external forces. This software evolution process is tightly coupled to the evolution of the* ecosystem *that the software system is embedded in.*

Remark 3.1.1
The research of software evolution, on a high level, can be split in two main directions:

- *Research that focuses on means to support a "good" evolution of a software system. This includes especially research on software development methodologies [192, 199, 230, 272, 281, 289] but also software design research (e.g., Gamma et al. [106]) and tools to improve software quality [59, 222, 233].*
- *Research that focuses on the analysis of software evolution. This includes the analysis of the evolution of a single software system [200, 201, 221] as well as the evolution of an entire ecosystem [113, 220] or analysis of software evolution itself [113, 200, 221, 225, 251].*

The present thesis attempts to contribute in both directions. Regarding the support of software evolution activities, we will present and evaluate approaches for software model completion. Edit operation mining and semantic lifting, presented in this thesis likewise, can support the analysis of the evolution of software models.

We next take a more formal look at evolution in general. We will discuss the link between patterns (i.e., observable concepts in software artifacts) and software evolution.

3.2 Towards a Formal Understanding of Evolution

In this section, we will try to establish a formal understanding of software evolution. In particular, complexity, as a central concept in the study of evolution, will be formally introduced and discussed. To have some common vocabulary and picture in mind, in section Section 3.2.1, we will introduce a simplified model of evolutionary dynamics motivated by our work on edit operation mining. This model is not intended to be a complete model of evolution. In particular, it can not explain selection. We will discuss the limitations in Section 3.2.3.

3.2.1 Assembly of Software Artifacts

Based on the concept of *building blocks* and the *combination* of building blocks to form new composite building blocks, we want to devise a simplified model for evolutionary dynamics. We will first discuss a few concrete examples and then generalize from these examples to a more abstract model.

Suppose we have a set of Lego bricks as building blocks. We can combine these bricks to build a new object by combining two (or more) bricks by putting (part of) the bottom of one brick on the studs of another brick. Recombining the bricks in this way, we can build new objects, for example, a house, a car, or a plane. There is usually not only one way to combine two or more bricks with each other, but many ways. Likewise, there is not only one way to combine the bricks to form a house, but many ways.

A more mathematical example is the assembly of functions.

Example 3.2.1 Assembly of functions.
Suppose we have a set of computable functions $\mathcal{F}$, realized by some language. We can then define the combination of two functions $f, g \in \mathcal{F}$ as the function $h = f \circ g$ (i.e., composition of functions), or more generally, we can define any computable function $\phi\colon \mathcal{F} \times \mathcal{F} \to \mathcal{F}$ as the combination of two functions. Starting with some subset $\mathcal{A} \subset \mathcal{F}$, we can then iteratively combine functions from $\mathcal{A}$ to build new functions (with possibly) increasing complexity. For more details on the assembly of functions, we refer to the work of Fontana [102].

If we abstract from details of the source code, we can consider a software system as a combination of the libraries used to build the software system.

Example 3.2.2 Assembly of software libraries.
For this example, we consider a contemporary package manager such as NPM for JavaScript, Maven for Java, or pip for Python. We can then package a set of software libraries in a package, and we can combine packages to form new packages. Compared to real-world software, this combination is quite abstract. In the real world, the software libraries will be used in the (source code of the) software system. That is, the concrete combination can become quite complex, compared to just considering the (set-level) combination of packages.

In this work, we will be concerned with the evolution of software models (e.g., UML models, EMF Ecore models, etc.). Therefore, central to this thesis is the assembly of software models, where the elementary building blocks are model elements, or, more generally, models are the building blocks themselves.

Example 3.2.3 Assembly of software models.
Two software models can be combined to form a new software model by merging the two (or more) models [44, 183, 211, 273, 285]. Alternatively, for the evolution of a single software model, we are typically interested in edit operations that are applied to the model. In the context of merging models, an edit operation could also be understood as a patch that is computed from the difference of two (or models)—as in our construction of edit operations in Section 2.4. More formally, to align the combination of software models with the formalization in Section 2.4, given a software model $m \in \mathcal{M}$, we can interpret an edit operation $\varepsilon \in \mathcal{E}$ also as a "minimal model" $m_\varepsilon \in \mathcal{M}$, where the preserved elements will match with the corresponding elements in the model m and there are no further elements in m_ε that are not involved in the application of the edit operation. This way, we can interpret the application of an edit operation $\varepsilon \in \mathcal{E}$ to a model $m \in \mathcal{M}$ as the two models $(m, m_\varepsilon) \in \mathcal{M} \times \mathcal{M}$ via a merge operation—assuming, as before, that application conditions are met and therefore no merge conflicts occur. That is, similar to function composition in Example 3.2.1, we can define a function $\phi\colon \mathcal{M} \times \mathcal{M} \to \mathcal{M}$ that combines two models. This construction yields another definition of edit operations by the identification $\varepsilon := \phi(m_\varepsilon, .)$, where ϕ will be realized by some merge operator.

To generalize from the examples seen above, we can define the assembly (of software artifacts) as follows:

Definition 3.2.1 Assembly.
Let $\mathcal{M}$ *be a* universe *of building blocks. The* combination *(also called* assembly*)*

of building blocks is a function $\phi\colon \mathcal{M} \times \mathcal{M} \to \mathcal{M}$ *that combines several building blocks to form a new building block.*

Remark
In general, it is also conceivable to combine more than two building blocks to form a new building block, that is, $\phi\colon 2^{\mathcal{M}} \to \mathcal{M}$*, where we denote the power set of* $\mathcal{M}$ *by* $2^{\mathcal{M}}$*. For most assemblies, the combination would be an iterative, sequential process. Furthermore, combinations of three or more building blocks could be modeled by successive applications of two building blocks, with some intermediate combined blocks*[a]*. We therefore prefer to restrict the definition two pairwise combinations. Other models of evolutionary dynamics are also conceivable, for example, Fontana [102] describes a model of evolutionary dynamics comprising a* universe*, an* interaction*, a* collision rule*, and a* system*. For the sake of simplicity, in the model introduced above, we did not distinguish between interaction and collision rule.*

a At least, we are not aware of any counterexample, in particular, in software engineering. Even in chemistry, reactions of three or more molecules typically happen in stages because it is highly unlikely that three molecules collide at the same time.

3.2.2 Measuring Complexity and Pattern Discovery

With the evolution model given in Section 3.2.1, given an initial family of building blocks (i.e., software models in our specific case), one can generate new building blocks by iteratively combining existing building blocks. In case the initial family includes models m_ε that belong to a generator S of edit operations (see Definition 2.4.7), by iteratively combining these models, we can reach every valid model in the universe $\mathcal{M}$. Anyway, as we can imagine, not every combination of building blocks will be meaningful. Let us consider two extreme cases (see also Figure 3.2):

Low complexity building blocks: On the one end, the universe $\mathcal{M}$ includes low complexity combined building blocks. These blocks have a low complexity (in a sense to be further specified below) and a high number of copies. For example, in software systems, we will very often have an abstraction for users that have a role in a system. We will find this concept (e.g., in the form of a table) in many software systems, that is, with a high copy number.

Purely random building blocks: On the other end, the evolution model given in Definition Section 3.2.1 allows for "random combination" that are of high complexity (in a sense to be further specified below). Of course, random combinations with a high complexity—for simplicity think of large random combinations of model elements such as classes—are rarely meaningful. For example,

in the upper right corner of Figure 3.2, a "physician" can have "airplanes". Even though a perfectly valid class diagram, a system that requires (multiple) relationship like this is less conceivable.

Most real-world software models consists of parts with a rather high complexity, but still have a high abundance (measured, for example in the number of system users, library usages, forks, et cetera). That is, these software models are somewhere "in between" the *low complexity/high abundance* case and *high complexity/low abundance* case. The evolutionary dynamics of this "real-world" evolution is depicted in the lower part of Figure 3.2. In some sense, the evolution dynamics seems to be biased compared to a purely random dynamics on the universe of building blocks (e.g., some uniform selection of available models for the merge).

One important consequence of this biased evolutionary dynamics is the occurrence of "patterns": when we observe a large collection of software models, we will find "complex" building blocks that occur more often than what we would expect by a random combination of building blocks.

To measure complexity, several related theories—Algorithmic Information Theory, Minimum Description Length, Minimum Message Length, and Assembly Theory, to only mention the most prominent ones—exist. We will give a short introduction to these measures:

Algorithmic Information Theory (AIT)
Algorithmic Information Theory [119, 204] is a computational framework that explores the fundamental relationship between data and the programs that generate them. At its core, Algorithmic Information Theory seeks to quantify the intrinsic complexity of objects by examining the length of the shortest program capable of producing them. Contrary to traditional information theory [255], which focuses on the statistical properties of data, Algorithmic Information Theory focuses on the intrinsic complexity of data without assuming they are derived by sampling from a probabilistic source.

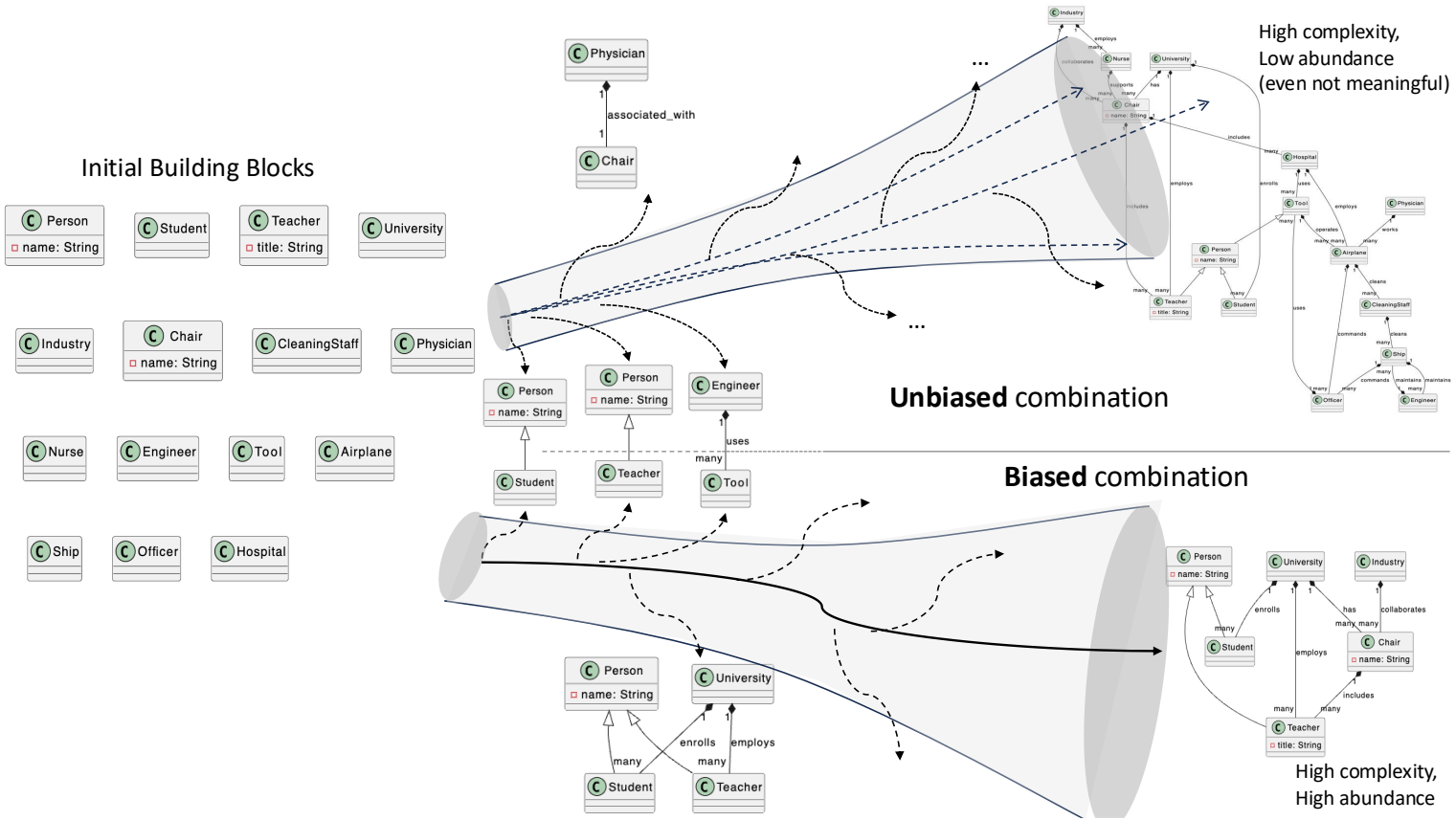

Figure 3.2: This graphics depicts possible outcomes through iterative combinations of classes (and associations) in the space of UML class diagrams. The initial building blocks are given by the classes on the left (and associations like inheritance, aggregation, and references). Iterative recombination according to Definition 3.2.1 leads to several possible future outcomes that are all syntactically correct class diagrams. Real-world systems are often of high complexity with a high abundance (copy number/number of users), as opposed, to very specific (i.e., simple) with high abundance, or random with high complexity but low abundance (adapted from Jaeger [144]).

Example 3.2.4
Suppose our data are given by a string:

$$s = \underbrace{000\ldots0}_{\text{1,000,000 zeros}}$$

While the string itself is long and seemingly complex, it can be generated by a very short (Python) program, such as:

```
print(1000000 * "0")
```

Interestingly, this string could also be the results of a fair coin flip, denoting heads as 0 and tails as 1. Without any further assumptions, this exact sequence is as likely as any other sequence of the same length. Of course, given the outcome, it would be very unlikely that the sequence is generated by a fair coin flip.

The main concept of Algorithmic Information Theory is the *Kolmogorov Complexity*:

Definition 3.2.2 Kolmogorov Complexity.
Let x be an object. The Kolmogorov Complexity *of x, denoted by $K(x)$, is the length of the shortest program that generates x.*

$$K(x) = \min_p \{|p| : U(p) = x\},$$

where $|p|$ is the length of program p, and U is a universal Turing machine (details can be found in any text book on algorithmic information theory [204]).

One might object that Kolmogorov Complexity is not well-defined, because this definition might be arbitrary due to several possible realizations of a universal Turing machine. Anyway, this arbitrariness is only up to a constant as one of the central theorems of Algorithmic Information Theory, the *Invariance Theorem*, shows:

Theorem 3.2.1 Invariance Theorem (informal).
Let U_1 and U_2 be different realizations of a universal Turing machine, then there a constant c, independent of x, such that

$$K_{U_1}(x) = K_{U_2}(x) + c \quad \forall x \in \{0,1\}^*,$$

where $K_{U_1}(x)$ and $K_{U_2}(x)$ are the Kolmogorov Complexities corresponding to U_1 and U_2, respectively.

Effectively, we can therefore choose any universal programming language (e.g., Java, Python, Rust, or C).

Example 3.2.5 Kolmogorov Complexity of $000\ldots0$.
In the example 3.2.4, given a string of k of zeros, the Python program would be of length $\log(k) + \log\log(k) + \mathcal{O}(1)$, representing k using some prefix-free code. The length of this program is much shorter than the string $000\ldots0$ itself. This illustrates how Algorithmic Information Theory distinguishes between the apparent complexity of data and its true algorithmic complexity, which depends on the length of the shortest generating program but is independent of any assumptions on a *prior* probability distribution.

There are many remarkable insights that can be derived within the framework of Algorithmic Information Theory. For example:

- Most of the strings $x \in \{0,1\}^*$ are incompressible in the sense that $K(x) \approx |x|$. These incompressible strings are also called *Kolmogorov random* or *Martin-Löf random*. This can easily be understood from the fact that, fixing $|x| = n$, only $\frac{1}{2^k}$ of the strings can be compressed to $K(x) < n - k$.
- The code given by K, that is, the shortest program encoding x, is an approximately optimal code (i.e., minimal expected code length, for large $|x|$), for any probability distribution P on $\{0,1\}^*$. In classical information theory, optimal codes are only defined depending on a probability distribution P.
- Many definitions, such as entropy, mutual information, etc. of (Shannon) information theory carry over to Algorithmic Information Theory.
- Using the language of Algorithmic Information Theory, *Occam's Razor* can be formulated in a mathematical precise sense, as we will see below.

Unfortunately, as we will elaborate on later, the Kolmogorov Complexity is not *computable*. Nevertheless, additional to its use in theoretical computer science, there are practical useful applications of the theory as well. One of these applications is in model selection. Before making this relationship more precise, we transfer the concept of Kolmogorov Complexity to the concept of assembly (see Section 3.2.1).

Example 3.2.6 Kolmogorov Complexity of Building Blocks.
We assume that the universe of building blocks $\mathcal{M}$ is countable (which is the case for all examples given in Section 3.2.1). We can then define an encoding of the building blocks in $\mathcal{M}$ by a prefix-free code: First, we enumerate the building blocks in $\mathcal{M}$. Then, we use some prefix-free code for the natural numbers to obtain a prefix-free code for the building blocks. Having a prefix-

free code, given a collection (also called database) of building blocks, we can define a code that encodes the collection by the concatenation of the codes of the building blocks in the collection. Based on this encoding, we can then define the Kolmogorov Complexity as before (see Definition 3.2.2).

More specifically, for software models, we can use a common encoding to define the Kolmogorov Complexity of a collection of software models.

Example 3.2.7 Kolmogorov Complexity of Software Models.
For the concrete case of software models, we typically have already a code that encodes the models. For example, for UML models, we can use the XMI format. We would then define some separator, and we can define codes for collections of models, similar to the more general definition of collections of building blocks above. Assume that we have a collection of all valid UML models, encoded as $x \in \{0,1\}^*$ as above. We can then compute the Kolmogorov Complexity $K(x)$ of the collection of UML models. As elaborated above, we would expect the collection to include high complexity, high abundance building blocks, that is, *patterns*. Because a pattern description can be reused, their high abundance suggest that x can be compressed to a much shorter code than $|x|$, that is, $K(x) \ll |x|$. On the other hand, if the collection includes many random building blocks (i.e., Martin-Löf random), we would expect $K(x) \approx |x|$. Anyway, it is unclear how the program p that generates x and minimizes $\{|p| : U(p) = x\}$ would look like. In fact, due to the uncomputability of the Kolmogorov Complexity, we can not explicitly compute $K(x)$, and we can not find the program p.

We see that the Kolmogorov Complexity can be used as a measure for complexity. Anyway, the Kolmogorov Complexity can not be computed and from Definition 3.2.2, it is unclear how to utilize Kolmogorov Complexity for pattern discovery. We will next introduce so-called *two-part codes*, that will then help us to define a more practical approximation of the Kolmogorov Complexity. One can show that interpreting p as $p = i'y$, where i' is a prefix-free encoding of an integer i and y is and input to some Turing machine, Definition 3.2.2 can be reformulated as

$$K(x) = \min_{i,y}\{|i| + |y| : U(i'y) = x\} \tag{3.1}$$

$$= \min_{i,y}\{K(i) + |y| : T_i(y) = x\} + \mathcal{O}(1), \tag{3.2}$$

where T_i is the i-th Turing machine—according to some enumeration of Turing machines—that computes x given input (also called program) y. The argument of the elements in the set in the last of these equations is often interpreted as a two-part

code: The first part $K(i)$ can be interpreted as the complexity of T_i as a *model*[8] that can be used to describe x, and the second part y describes how the model T_i has to be used to describe x and further specifics of it that can not directly be explained by the model T_i.

Example 3.2.8 Regression Analysis.
As a classical example assume that x encodes n points in the Euclidean plane. The first part i, for example, could then encode a Turing machine T_i that describes a polynomial to fit the n points. The second part y would then encode the residuals (or errors) of the polynomial fit. In this case, we would especially be interested in a "good enough" but not too complex model T_i (i.e., of not too large degree and acceptable accuracy of coefficients of the polynomial) that can be used to describe the n points. In machine learning, the balancing of complexity and goodness of fit is often referred to as the *bias-variance trade-off*. *Model selection* is the process of finding an optimal model from a given *class of models*.

We can also apply this construction to less numerical examples in software engineering, as the following example demonstrates:

Example 3.2.9 Python Ecosystem.
As another example, consider the Python ecosystem (similar to Example 3.2.2). In this case, x would be a collection of (all) Python projects. The first part i could then encode a Turing machine T_i that describes the Python language as well as all libraries and packages that exist in the Python ecosystem at a certain point in time. The second part y would then only encode the specific assembly of libraries in the python projects, that is, new projects with no further regularities. Note that, in general, for this to be optimal, the Turing machine T_i needs to be a maximally compressed description of the Python ecosystem. Furthermore, the description of the program y also needs to be as compressed as possible, that is, without further regularities or "reuse opportunities". That is, for i and y to minimize Equation 3.2, y would very likely not be an ASCII encoding of the source code of the project.

For this thesis, we are interested in the assembly of software models:

Example 3.2.10 Software Models.
Similar to the previous example, we can also apply the concept of two-part codes to the assembly of software models 3.2.3. More concretely, in the case of

8 Not to be confused with the software models we are concerned with in this work.

UML models, the first part i could encode a Turing machine T_i that describes the UML language with all its constraints and usage patterns. The second part y would then encode the specific assembly of models, i.e., the concrete UML models without any further regularities. As in the previous example, for this to be optimal, the Turing machine T_i needs to be a maximally compressed description of the UML language and y would be a minimal description of the models, given the "ecosystem" described by T_i.

As these previous examples demonstrate, having a two-part code allows us to distinguish between a *model* part (the T_i part) and a *residual* part (the y part). In typical machine learning applications, the *residual* part is often related to *noise*, that is, a prediction error that can not be explained by any model (in the model class). The latter two examples also show that it might not be trivial to choose a Turing machine T_i and the input y in a way that T_i also describes "meaningful" regularities. Many combinations in the argument of the minimization in Equation 3.2 are conceivable.

Based on Equation 3.2 and ideas from Kolmogorov, a mathematical rigorous theory for model selection has been developed [332]: For simplicity, the class of Turing machines in Equation 3.2 is restricted to Turing machines that enumerate a set H[9] of finite strings and given input y output the y-th string in H, if $y < |H|$, and a special `undefined` string otherwise. For the specific case of "assembly" considered here, the limitation to finite strings x is not a limitation, because we do only consider finite programs or software models. Indeed, even the set of all existing software models has a finite description (and the same is true for the set of all existing source code).

Definition 3.2.3 Two-Part Code (set version).
Let H be a set of finite strings. A two-part code *for x is a pair (i, y), where i describes a Turing machine that enumerates H, $x \in H$, and y describes a program to select $x \in H$. As usual, the complexity of H is denoted by $K(H)$ and the program corresponding to this shortest description is denoted by H^*. To compute x from H, a minimal complexity of $K(x|H)$ is required. Obviously, for finite H,*

$$K(x|H) \leq \log |H| + \mathcal{O}(1), \tag{3.3}$$

because we can define the index of x in H by a self-delimiting code of length $\lceil \log |H| \rceil$.

9 H stands for hypotheses because this can also be seen as a set of hypotheses.

Because, for a finite set H, there is a program that first computes H and from H computes x, we have the relation

$$K(x) \leq K(H) + K(x|H) + \mathcal{O}(1) \tag{3.4}$$
$$\leq K(H) + \log|H| + \mathcal{O}(1), \tag{3.5}$$

for the set version of the two-part code and the Kolmogorov Complexity of a binary string x.

There are three important observations to be made with respect to the selection of the model H:

1. An element $x \in H$ is called *typical element*, if equality holds in Equation 3.3 (up to some constant). If this is not the case for x, then there is some structure in x that is not fully captured by H. In that sense, in order to have a "good" model H, we want H to be such that x is a typical element.

2. A good two part code should be close to optimal, that is, we also want equality in Equation 3.4. A set H for which x is typical, and we have equality in Equation 3.4 is called *optimal*. Given a binary string x, a shortest description H^* for an optimal set H is called an *algorithmic sufficient statistic* for x.

3. The set $H = \{x\}$ is always optimal for x as it is clearly typical ($K(x|\{x\}) = \mathcal{O}(1)$), and also

$$\begin{aligned} K(H) + K(x|H) + \mathcal{O}(1) &= K(\{x\}) + K(x|\{x\}) + \mathcal{O}(1) \\ &= K(\{x\}) + \mathcal{O}(1) \\ &= K(x) + \mathcal{O}(1), \end{aligned}$$

since $K(\{x\}) = K(x) + \mathcal{O}(1)$ (only some constant overhead is added for computing a representation for $\{x\}$ from x). This shows that the criterion to be optimal is not strong enough for H to be a "good" model—in the sense that it captures "meaningful" regularities of x—for the binary string x. To fix this, one can introduce a parameter α and require that $K(H) < \alpha$. Then, increasing α from $\alpha = 0$ to $\alpha = K(\{x\})$, one might find an optimal set H with $\alpha < K(\{x\})$. An optimal set H for which *alpha* is minimal is called an *algorithmic minimal sufficient statistic* for x (see Figure 3.3). If no optimal set other than $\{x\}$ exists, x is Kolmogorov random.

Let us summarize these three observations by following model selection principle:

Principle

Principle 3.2.1 (Minimum Description Length formulation of Occam's Razor)**.** Suppose we have binary data x, a family of models $\mathcal{H}$, and two models (i.e., our set H above) $H_1, H_2 \in \mathcal{H}$ that we want to compare. If

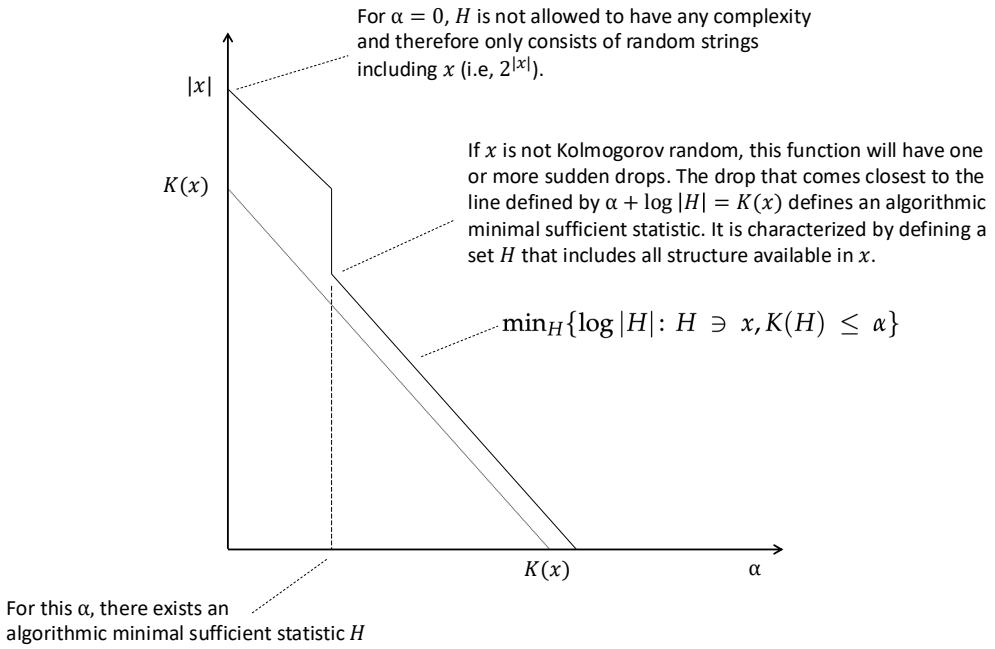

Figure 3.3: In 1974, Kolmogorov (and later others [332]) has introduced a *structure function* $h_x(\alpha) = \min_H\{\log|H|: H \ni x, K(H) \leq \alpha\}$. At the minimal $\alpha < K(\{x\})$ that admits an optimal set H, the graph of this structure function will have a "drop". The drop that comes closest to the line defined by $\alpha + \log|H| = K(x)$ defines a set H that includes "all structure available in x"—the algorithmic minimal sufficient statistic.

- $K(H_1) < K(H_2)$,
- $\log|H_1| - K(x|H_1) \leq \log|H_2| - K(x|H_2)$, and
- $\log|H_1| + K(H_1) \leq \log|H_2| + K(H_2)$,

then H_1 is preferred over H_2.

That is, by this formulation of Occam's Razor, a good model has low complexity, explains a lot of structure in the data, and still admits a short description of the data (i.e., allows for a large loss-less compression).

Example Library Usage Patterns.
As an example, consider the Python ecosystem again. Suppose, we want to bundle several libraries and reuse these bundles instead of the individual libraries within projects. Let x be a binary encoding of the set of all libraries used by all Python projects. Each entry in the set corresponds to the description of libraries used in one project. The candidate models $H \in \mathcal{H}$ would then

be sets H that contain sets of libraries. A Turing machine T_i could employ the fact that libraries are often used in combination in "bundles", that is, the libraries are not used independently of each other. To generate the description of the libraries used by a single project, T_i could then just combine several of these bundles of libraries (i.e., a union of the sets of libraries used in the bundles). A good model H would be such that the generated project descriptions are typical, that is, the libraries used in the projects are often used in bundles. Furthermore, in order to minimize $K(H)$ the number of bundles should also be small. And finally, the total description length should be minimized. Effectively, T_i would leverage library usage patterns to generate a minimal description of the Python ecosystem.

In Example 3.2.10, we have seen a two part code for software models. We could now ask for a minimal description of a generating Turing machine T_i, constrained to the idea that the UML models are assembled from certain patterns. To find the maximal compressed description of a set of UML models T_i (or at least an approximation to it), we can apply the same principle as in the previous example.

Example Patterns in UML Models.
The description of the set of UML models T_i can employ the fact that certain combinations of model elements are more likely to occur in UML models than others. A single UML model can then be generated by T_i by combining these typical combinations of model elements, that is, *patterns*. T_i could then utilize these combinations, as well as their probability (given some context) to generate (i.e., assemble) UML models. Again, a good model H would be such that the generated UML models are typical, the description of all reused patterns and a description for their assembly is minimal, and the total description length is optimized. Figure 3.2 schematically shows that real-world evolution will lead to a bias on the universe of all valid models. It is exactly this bias that a good model H or its generating Turing machine T_i should employ.

These examples might seem to be inefficient in practice, because we are basically enumerating all possible models and our sets H are very large. We will next discuss a more practical approach to measuring complexity.

Minimum Description Length Principle (MDL)
In the formulation of Occam's Razor (see Principle 3.2.1) the terms $K(H)$ and $K(x|H)$ can only be approximated. Finding an (overall) optimal set H for infinite x is not possible[10] and, for finite x, not feasible. Also, the theory above is "asymptotic" in

10 Even if one has an optimal set, one doesn't know that it is the optimal one.

the sense that it is only valid for large x (when the constant $\mathcal{O}(1)$ really becomes negligible). Motivated by Algorithmic Information Theory, Rissanen [266] introduced the *Minimum Description Length Principle* as a practical approach to model selection. The Occam's Razor principle based on Algorithmic Minimal Sufficient Statistics is often seen as *ideal MDL*. In practice one typically starts with a set of initial hypotheses $\mathcal{H}$ for which an optimal encoding is known or can be computed. Typically, these hypotheses are given by probability distributions over a space including x (e.g., $\Sigma := \{0,1\}^*$). The principle task (i.e., to select the hypothesis that minimizes the description length) is the same for both the ideal MDL and the *practical MDL*. A difference often encountered in practical MDL is, that one is interested in a good encoding for all x in some space, not only for a single observed x. That is, as in Bayesian approaches, one assumes that the data x is generated by a probabilistic source. This is a more conceptual than a methodical difference, because it can be shown that Kolmogorov Complexity (asymptotically) has a universal property, that is, that it leads to (almost) optimal encoding, even without knowing the exact outcome x beforehand.

Several *universal models*, that is, hypotheses in the form of probability distributions $\mathcal{H}$ that lead to (almost) optimal description length for all possible observations, have been proposed. Examples of models used in Minimum Description Length are Normalized Maximum Likelihood, Bayesian Universal Model, and also two-part codes. For example suppose we have some parametrized probability distribution

$$\mathcal{H} = \{P(\cdot|\theta) \colon \theta \in I \subset \mathbb{R}\}$$

and some previous assumption or prior $W(\theta)$ (i.e., a probability distribution on I). In the Bayesian Universal Model, one then computes a Bayesian mixture

$$P_{Bayes} := \int_I P(\cdot|\theta) W(\theta) d\theta.$$

It can be shown that the code defined by P_{Bayes} is optimal in a sense that code length is even smaller than a two-part code that has a part to transmit the selected model (i.e., the parameter θ) and then use $P(\cdot|\theta)$ to encode the data[11].

Usually the universal models make use of a probabilistic *prior* (e.g., the W prior we used above) distribution on the distributions. In ideal MDL—based on Kolmogorov Complexity—this prior would then be given by the algorithmic probability

$$P(x) = 2^{-K(x)}.$$

Therefore, Minimum Description Length can be seen as a practical realization of our formulation of Occam's Razor in Principle 3.2.1. Minimum Description Length

11 It is known from classical information theory by the so-called *Kraft's inequality* that probability distributions P lead to prefix codes that minimize the average code length—assuming the data is distributed according to P.

has become a rich field of research and many priors have been proposed, and their guarantees have been understood. A good introduction to the topic is given by Grünwald [117, 118].

Anyway, in many cases no universal prior is known or can be computed. For example, in this thesis, we do not have a universal prior for software models (or labeled graphs with possibly infinite label alphabet). Furthermore, we believe that the purely probabilistic viewpoint does not reflect the domain of software models well. Indeed, software models are not generated by a probabilistic source, but by human design. For example, the interpretation of the "incompressible" part of a software model as noise is not appropriate. Instead, we believe that to some extent this "incompressible" part rather reflects a lack of information (e.g., about the market, the users, or requirements).

Anyway, Minimum Description Length is often used in a *crude* form [117, 118]. In crude MDL, one uses a two-part code to encode the data and the model:

$$L(D) = \min_{\mathcal{H}}\{L(D|H) + L(H)\},$$

where $L(D)$ is the length of the encoding of the data, $L(H)$ is the length of the encoding of the model, and $L(D|H)$ is the length of the encoding of the data given the model. The selection of the model is typically done in an *ad-hoc* manner. It is known that this crude MDL is often not optimal because the encoding of the model is often arbitrary. It is frequently used in practice, because of its simplicity and often good performance in practice. Still, using a crude MDL approach, care should be taken to avoid overfitting—or missing regularities likewise.

Example 3.2.11 Patterns in UML Models.
In the context of UML models, as above, we assume that patterns are present in the UML models. Roughly speaking, this means that there are certain subgraphs that appear more often in graph representation of the models than what one would expect by chance (i.e., if all combinations that are valid according to the UML meta-model are equally likely). The space of models would then consist of a codebook for patterns and a code for how these patterns are assembled (i.e., glued together) to form the UML models. The description of the data in terms of this model is then a minimal description utilizing the (encoded) assembly procedure. Utilizing this approach to pattern mining, a probability distribution for x is not necessary. Anyway, in the encoding of the codebook one would take the complexities and number of occurrences (effectively the relative probabilities of a pattern) into account to minimize the expected description length of the references to the pattern in the description of the data. In general for graph pattern mining, several approaches based on Minimum Description Length have been proposed (e.g., GraphMDL [28], Subdue [83], or the approach presented in Section 6.2).

A more detailed discussion of this approach to pattern is presented in Section 3.5 and will be empirically evaluated in Chapter 6. Similar to the example given above, in product-line engineering, Minimum Description Length can be leveraged to extract a product line from a set of products, by using features as reusable building blocks and the collection products as the data, as we will discuss later in this thesis (see Section 4.4).

Remark 3.2.1
We have introduced and discussed the Minimum Description Length principle in the context of software model evolution and pattern discovery. Anyway, this principle has many occurrences and practical applications. For example, Minimum Description Length can be used to derive tree-like taxonomies from graph like information (e.g., from a concept lattice), by encoding the graph as a tree plus the cross-tree relationships and then minimizing the description of this decomposition. Similarly, one could reverse engineer software components by describing the dependency graph as connected modules and the modules with their internal dependency graphs (i.e., one would decompose the (sparse) dependency matrix, to compress its description). Minimum Description Length-based approaches to graph summarization, such as one by LeFevre et al. [198] could be used for this purpose.
In general Minimum Description Length is an approach which can come to the rescue whenever one has to deal with a decomposition of a complex structure into simpler parts, which is often the case in Software Engineering.

Minimum Message Length Principle (MML): While both, *Minimum Message Length* and Minimum Description Length, aim to balance model complexity with goodness of fit via minimizing the description length, they differ in some aspects. Minimum Message Length is explicitly formulated from a Bayesian perspective and focuses on fully specified models, while Minimum Description Length tends to be more focused on model classes and avoids explicit use of priors. Optimization in Minimum Message Length is usually done by minimize average code length, while Minimum Description Length is often used to minimize the worst-case code length. Furthermore, Minimum Message Length solely uses two-part codes, while Minimum Description Length can use a variety of coding schemes. However, in practice the two approaches often lead to similar results [117]. We mention Minimum Message Length here only for completeness and want to refer to the literature for further details [336].

Remark 3.2.2
From a conceptual standpoint, Algorithmic Information Theory aligns more closely with the objectives of this work than the Minimum Description Length Principle or the Minimum Message Length Principle. The primary reason for this alignment is our

aim to describe our observations—a collection of software models—in a manner that captures their inherent regularities. This focus on describing a single observation in a minimalistic way is precisely the goal of Algorithmic Information Theory. In contrast, both Minimum Description Length and MML adopt a more probabilistic perspective on the data and the models, assuming that observations are generated by a probabilistic source. This probabilistic framework necessitates making assumptions, often in the form of priors, about the space from which the observations are drawn. In a practical setting, the three approaches, though conceptually different, seem to align more closely. For example, since we can not compute $K(H)$ for infinite sets H, we have to make some assumptions about the space of models H which brings us to a space $\mathcal{H}$ in Minimum Description Length. As another example, its egregious that an optimal encoding (in Algorithmic Information Theory) for a single x turns out to be almost optimal for any $x \in \Sigma = \{0,1\}^*$—independent of the probability distribution on Σ—in an asymptotic sense.
For the use cases studied in this thesis, we can use the knowledge that our data will be generated by a human design process as a kind of "prior" to design the model space $\mathcal{H}$ in a way that it captures the regularities in the data. In particular, for the design of software models, the modeling tool offers edit operations that the modeler can use. They leave their traces in the software model repositories that we can observe.

Assembly Theory (AT)
Other than Algorithmic Information Theory, Minimum Description Length Principle, or Minimum Message Length Principle, *Assembly Theory* focuses on the description of the *assembly* of objects based on a complexity measure—the *assembly index*. In Assembly Theory, an *object* is not an atomic building block but "[...] the constraints to construct it from elementary building blocks is quantifiable" [295]. What is meant by this formulation is that from every object, it is possible to derive some of its building blocks. Since these building blocks themselves might not be atomic (even if we do not know so yet), we can further derive building blocks from these building blocks, and so on.

Definition 3.2.4 Assembly Index, Copy Number.
Let i be an object that can be assembled from building blocks. The assembly index a_i of object i is then the length of the shortest assembly pathway *in the* assembly space *(i.e., all possible assemblies) of the object. Given an ensemble I (i.e., a set) of objects, the* copy number *n_i of an object i is the number of occurrences of the object in the ensemble I.*

Example 3.2.12 Assembly of functions.
In example 3.2.1, we have defined assembly of functions as the composition of functions. The objects of assembly theory in this example would then be the functions. Suppose we are given the function $f(x) = sin(a * x + b) + cos(a * x + b)$. Then, with $f_1(x) = a * x + b$, $f_2(x) = sin(x)$, and $f_3(x) = cos(x)$, we have an assembly pathway of length 5 since $f(x) = f_3(f_1(x)) + f_2(f_1(x))$, from the basic building blocks $+$, $*$, cos, and sin, saving the recomputation of $f_1(x)$. For the assembly index we then have $a_i = 5$.
Most likely, in the Algorithmic Information Theory viewpoint, we would have a similar reuse of existing functions. Anyway, Algorithmic Information Theory could also detect even shorter formulations that are not bound to any initial building blocks and the general composition assembly.

The theory furthermore defines a quantity called *assembly* A ("[...] total amount of selection necessary [...]" [295]) for an ensemble I of objects, which is defined as

$$A = \sum_{i=1}^{N} e^{a_i} \left(\frac{n_i - 1}{N_T} \right), \tag{3.6}$$

where a_i is the assembly index, n_i is the copy number of object i, N is the total number of unique objects in the ensemble, and N_T is the total number of objects in the ensemble. As we will later demonstrate, the measure A is compression measure that is related to the amount of information that can be saved in the description of the ensemble I by reusing objects.

Assembly Theory focuses on the description of the complexity of single objects —molecules, in particular. Complexity is used for the primary goal of detecting selection and life-derived objects.

What is not directly included in the previously discussed complexity measures (i.e., Kolmogorov complexity and description length) is the *recursive assembly* of objects (as assembly pathways), though this recursive assembly can be encoded in candidate models $\mathcal{H}$ used, for example, as part of an Minimum Description Length approach. Kolmogorov complexity, will, at least, "leverage" reuse of building blocks along an assembly pathway, but it might find even shorter descriptions that compress the object with some "mechanism" that goes beyond assembly. In this sense, the assembly index is only an approximation to the Kolmogorov complexity of an object. One might even want to exclude possible shortcuts to complexity and can therefore interpret the assembly index as a measure of the complexity along assembly pathways.

In our work, we are interested in the identification of patterns (i.e., "meaningful" substructures) within repositories of software models. Even though this application is not directly covered by Assembly Theory, the assembly pathway of objects (in our case software models) can be seen as a sequence of patterns that are assembled to form the software models.

Another interesting terminology in Assembly Theory is that of *assembly universe, assembly possible, assembly contingent, and assembly observed* [295]. Sharma et al. [295] define the *assembly universe* as the set of all possible objects that can be assembled from a given set of building blocks. For example, for software models, the meta-model of the modeling language would define the assembly universe. Similarly, for source code, the syntax of the programming language would define the assembly universe. Usually there are some additional constraints that limit the assembly universe, giving the *assembly possible*. In the case of software models, these could be additional constraints not directly encoded in the meta-model, such as constraints given by an additional constraint language (e.g., OCL in the UML context). For source code, these could be constraints given by the compiler or even runtime constraints (e.g., memory constraints). The *assembly contingent* is then the set of objects that can be assembled from the building blocks that are available at a certain point in time. Software models would be assembled from the building blocks that are available in the modeling tool or certain pre-defined components are reused (e.g., in Simulink). For source code, we reuse libraries or frameworks that are available at a certain point in time [189]. Indeed, we do not want to reinvent the wheel but instead reuse existing software components. Finally, assembly observed is the set of objects that are actually assembled from the building blocks that are available at a certain point in time.

Assembly Theory basically states that if assembly observed differs from assembly contingent, then selection has taken place. That is, the observations of assembly that can not purely be explained by the laws imposed on the assembly universe are due to selection[12]. The growth dynamics (i.e., number of distinct objects) of the assembly universe is usually super exponential, while the growth dynamics of the assembly observed can be sub-exponential. We will usually (as we have indicated in Figure 3.2), rather have a bias towards increasing the copy number of some combinations, while other possible combinations never come into existence.

Remark 3.2.3
Assembly Theory is a simplification and, contrary to the paper title "Assembly theory explains and quantifies selection and evolution", fails to explain selection [1, 144]. Anyway, it provides an interesting and simplified framework to discuss evolution of objects and does not focus on a particular universe where these objects live in.

12 In Assembly Theory, there are two selection mechanism. One is on the amount of less complex (i.e., "older") objects that are reused in the future and one is on the combinations that actually are assembled in the assembly observed.

3.2.3 Limitations of Simplified Models of Evolution

Given the tools and concepts introduced in Section 3.2.1 and Section 3.2.2, there are still some limitation in the description and study of the evolution of software models. We will outline the most important ones here.

Uncomputability of Kolmogorov complexity As we have seen, Kolmogorov complexity is not computable. Algorithmic Information Theory provides some important insights into the nature of complexity and it should be considered rather as a theoretical foundation of complexity than a practical tool for measuring it.

Algorithmic Information Theory is not evolution theory As a theory about the information content and the randomness of data, Algorithmic Information Theory has no innate connection to the evolution of objects. It is conceivable that a theory of evolution could be built on top of Algorithmic Information Theory, but again, this would be a theoretical foundation rather than a practical tool. This and the previous point are the reason to look into methodologies that, on the one-hand, are computable, and, on the other hand, are more closely related to the evolution of objects—and software models, in particular.

Missing side effects and Interface Variability Regarding more practical description length approaches to pattern discovery, the question arises if the (restricted) set of models (or hypotheses) (i.e., the set $\mathcal{H}$ used above) can capture sufficient aspects of the evolution. Indeed, it can never be completed as Kauffman [158] argues, because for this also (all) side effects would need to be part of the model. This kind of brings us back to the more complete but infeasible Kolmogorov complexity viewpoint. What we can hope for is that after fixing the current point in time, the set of hypotheses is sufficiently expressive to capture the regularities in the data that are "meaningful". We just need to give up the idea that the same set of hypotheses can be used to predict the future. One observation that highlights this limitation is the emergence of new interfaces. Assembly theory, for example, does not consider interface variability. That is, how to the building blocks of the assembly can be combined to form new interfaces. This is particularly important in the study of effects like emergence. For example, with increasing complexity, a new interface mechanism can emerge. For example, when calling a function on a remote server, one typically doesn't care about the binary representation of the function call but rather about the JSON object that is sent to the server. The lower-level underlying mechanisms (e.g., error correction codes, physical transmission) are abstracted away. Still, they are a necessary function without which the higher-level interface would not work. Assembly theory assumes a fixed assembly universe and similarly Minimum Description Length and Minimum Message Length typically fix the set of hypotheses (also called models) $\mathcal{H}$.

Luckily, in the context of software models, a given meta-model constrains the set of possible models[13]. Within the models (i.e., excluding the effect the models have on the environment or the interaction with the environment, in general), we therefore do not have to worry about this limitation too much.

Natural Ecosystems vs. Sociotechnical Ecosystem No precise link between software evolution (i.e., an evolution in/of the "technosphere" [229]) and natural evolution (i.e., an evolution in/of the "biosphere")—to the best of our knowledge—has yet been established. Especially, software and systems are typically *designed* by humans for a specific purpose, while natural objects evolve due to natural selection. Anyway, not everything that has been designed will also be *adopted* by users or developers. It is conceivable that it is exactly the adoption of a particular software (or function) that corresponds to the selection we know from natural evolution. In this sense, we believe that patterns in software ecosystems are related to selection, biased evolution, and (the emergence of new) function.

Limitations of statistical complexity measures Some authors [1, 144] argue that the assembly index and statistical complexity measures are not sufficient to explain selection. For example, a bias can also be due to founder effects, that is, if we draw a small sample from some population there might be a bias (i.e., selection bias but in a random sense) and this bias might persist (e.g., through reproduction). Furthermore, Hector Zenil[14] argues that statistical complexity measures based on counting exact copies are more likely be related to "[...] generative mechanisms of crystals and not life." Hazen et al. [125] show that pathway complexity measures[15] can also be large for non-biological (e.g., mineral) systems. If similar measures are applied to software models, it is therefore imperative to empirically validate these measures.

Look into the past vs. look into the future Even more critically, we can raise the question if evolution is based on the combination of building blocks at all—or, a bit less radical, if the combination of building blocks and the knowledge of these building blocks and patterns is a necessary ingredient of a proper evolution theory. Indeed, as autoregressive data-driven models (most prominently large language models) have demonstrated, some future states can be predicted from the past context, without *explicitly* knowing the building blocks or patterns. Though we do not know yet about the merits and limitations of these models for software evolution, their preliminary success in the field of software engineering raises the question if we need explicit pattern knowledge

13 Since also meta-models can evolve [126, 227, 271], this statement has to be taken with a grain of salt.

14 https://manlius.substack.com/p/whats-going-on-with-assembly-theory

15 Marshall et al. [212] link life to high assembly indices, which is surprising since they ignore the copy number in this hypothesis, which they postulate is related to selection in the follow-up work by Sharma et al. [295].

at all to support the evolution of software or if patterns are rather descriptive in nature.

3.3 Graph Mining

In this section we provide a brief overview of graph mining and graph machine learning. As we have seen in Section 2.3, software models can be represented by graphs. Graph mining techniques can be used to extract pattern knowledge from graph-like data and therefore for pattern mining and learning pattern knowledge in model repositories. In Section 3.5, we then see the relationship between patterns and software model evolution.

Definition 3.3.1 Graph Mining, Graph Generation, Graph Machine Learning.
Graph mining *is a specialized area of data mining that focuses on extracting useful information and patterns from (a set of) graph-structured data.*
Graph generation *is the task of generating graphs that are similar to a given set of graphs, that satisfy certain properties, or that follow specific rules.*
Graph machine learning *concerns the application of machine learning techniques to graph-structured data.*

Examples for graph mining tasks include frequent subgraph mining [151, 242, 355], graph classification [176], graph clustering [6], graph similarity [360], link prediction [216], or anomaly detection [12]. A general introduction to graph mining is given by in the book by Cook et al. [72].

Input to graph mining algorithms can be a single graph or a collection (or a set) of graphs. We will call the latter a *graph database*, and denote it by D. For some graph mining tasks, for example, graph classification, additional information is provided in the form of labels or classes or meta-data for the graphs as part of the input data set. For this work, we will only focus on (frequent) subgraph mining, which essentially is an *unsupervised* learning approach, and consequently does not require additional information.

3.3.1 Frequent Subgraph Mining

Frequent subgraph mining concerns the identification of subgraphs (cf. Section 2.2) that occur frequently in a graph database. For example, applied to simple change graphs (see Definition 2.4.4), frequent subgraphs would correspond to changes that have frequently occurred together in the past.

Definition 3.3.2 Frequent Subgraph Mining.
Let D be a database (i.e., a set) of graphs. A subgraph G is frequent *in D if the number of graphs in D that contain G (see Definition 2.2.3) is greater than a threshold $t \in \mathbb{N}$. The* support *of G, $\text{supp}_D(G)$, is either the number of graphs in D that contain G (for* transaction-based *subgraph mining), or the number of occurrences of G in D (for* occurrence-based *subgraph mining).*

There are several dimensions along which several "flavors" of frequent subgraph mining can be distinguished:

Directed vs. undirected graphs: In directed graphs, edges have a direction, while in undirected graphs, edges are bidirectional.

Labeled vs. unlabeled graphs: In labeled graphs, nodes and edges have labels, while in unlabeled graphs, nodes and edges are not labeled.

Weighted vs. unweighted graphs: In weighted graphs, nodes and edges have weights, while in unweighted graphs, nodes and edges do not have weights.

Subgraph definition: Induced subgraphs (i.e., all edges between nodes in the supergraph are also part of the subgraph), embedded subgraphs (i.e., all edges between nodes in the supergraph are not necessarily part of the subgraph), or overlapping vs. non-overlapping (i.e., whether different subgraphs are allowed to share nodes and edges in the supergraph, or if nodes and edges can only be used for one subgraph that will be counted).

Transaction-based vs. occurrence-based: In transaction-based subgraph mining, a subgraph is counted only once per graph in the database, while in occurrence-based subgraph mining, a subgraph is counted every time it occurs in a graph in the database.

Closed graph: A subgraph is *closed* if there is no supergraph with the same support.

Constraints: Additional constraints can be imposed on the subgraphs, such as the number of nodes, the number of edges, the number of connected components, or the number of occurrences of a certain label, etc.

Example 3.3.1 Frequent Subgraph Mining.
Consider the graph database D in Figure 3.4. The database contains three undirected graphs G_1, G_2, and G_3. Node colors indicate the node label and the edges are neither labeled, nor weighted. In this example, we selected three subgraphs g_1, g_2, and g_3 to illustrate the concept of frequent subgraph mining. With a threshold of $t = 2$, all of them are frequent in D.

Subgraph g_1 has a support of $\text{supp}_D(g_1) = 3$, independent of the flavor. It appears in all three graphs in D, and only once per graph. The graph g_1 is also a closed graph in D, since no supergraph has the same support.
The support of g_2 depends on the flavor. Indeed, for transaction-based frequent subgraph mining, it only occurs in G_1 and G_3, and thus has a support of $\text{supp}_D(g_2) = 2$. For occurrence-based frequent subgraph mining, the support depends on whether overlaps are allowed or not. If overlaps are allowed, we have $\text{supp}_D(g_2) = 4$ as it occurs three times in G_3 and once in G_1. If overlaps are not allowed, we have $\text{supp}_D(g_2) = 2$ as it occurs once in G_1 and once in G_3. The graph g_2 is not a closed graph in D (for occurrence-based frequent subgraph mining), because it can be extended by attaching a green node to the gray node, without decreasing the support. The subgraph g_3 is frequent in D with a support of $\text{supp}_D(g_3) = 2$, again independent of the flavor. It is not a closed graph in D, as it can be extended to a graph that isomorphic to G_2 with the same support.

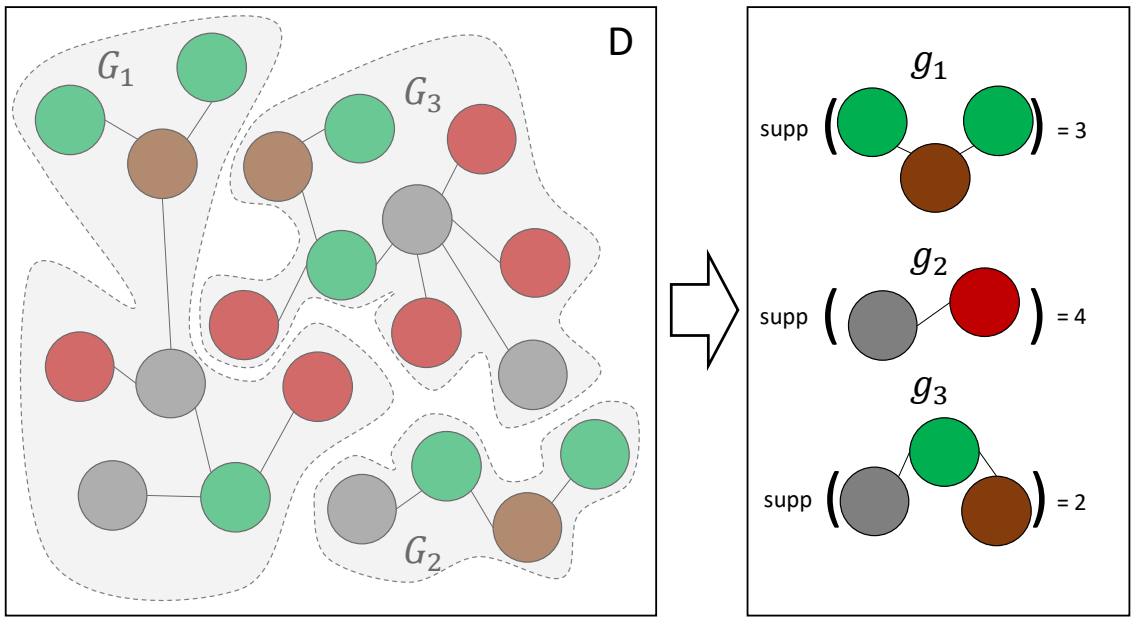

Figure 3.4: Frequency of subgraphs within a graph database $D = \{G_1, G_2, G_3\}$. A sample of subgraphs with their occurence-based support is shown.

Several approaches [151, 242, 355, 356] have been proposed to solve the frequent subgraph mining problem *exactly* (as opposed to approximately). The most common approaches include

apriori-based approaches, which use the apriori principle to reduce the search space—pruning infrequent subgraphs early in the search process—in a breath-first search,

pattern-growth approaches, which grow patterns from smaller patterns by adding edges or nodes, and

depth-first search-based approaches, which use a depth-first search to explore the search space. Usually depth-first search-based approaches complement a pattern-growth approach.

We will not go into detail about these approaches, as they are well documented in the literature [151]. Instead, we only focus on the pattern-growth approach for transaction-based frequent subgraph mining, which is also the basis for one of the main contributions of this work (see Chapter 6).

Pattern-growth approaches typically start with frequent single nodes and maintain embeddings of the subgraphs in the database. Iteratively, they grow the subgraphs by adding edges or nodes, and check the support of the extended subgraph. For transaction-based approaches, the downward closure property can be exploited: if a subgraph is not frequent, none of its supergraphs can be frequent either.[16]

Pattern-growth approaches make use of several techniques to reduce the size of the search space:

Pruning stops the extension of a subgraph if certain conditions are met, for example, if the subgraph is not frequent anymore.

Redundancy elimination avoids the enumeration of subgraph candidates that are duplicates of other subgraph candidates.

Subgraph merging generates new candidate subgraphs by merging two existing frequent patterns instead of extending them one edge at a time.

Example 3.3.2 Pattern-growth approach.
Consider the graph database D in Figure 3.4. Assume we want to run a pattern-growth approach for transaction-based frequent subgraph mining with a threshold of $t = 3$. Let us focus on the brown node. It is part of G_1, G_2, and G_3, and thus is a frequent single node with a support of $\text{supp}_D(\text{brown node}) = 3$. Since the support of the subgraph consisting only of the brown node is $3 >= t$, we can extend the subgraph. From G_3, we see that we can only extend the subgraph by adding a green node, as with red, brown, or gray nodes the support would drop below $t = 3$. With the green node, we still have a support of 3, and can extend the subgraph further. From G_2, we see that we can only

16 This does not hold for occurrence-based subgraph mining, when overlaps are allowed.

add another green node to the brown node. The gray node attached to the brown node in G_1 is not present in G_2 and G_3. At this point, for every further extension, the support will drop below $t = 3$, and we consequently can stop the extension of the subgraph at this point.

As one can image from this example, the search space can grow exponentially with the size of the subgraphs. Indeed, we have the following result:

Theorem 3.3.1 Complexity of Frequent Subgraph Mining.
The problem of frequent subgraph mining is $\mathcal{NP}$-hard.

A proof of this result can be found in the work by Garey et al. [107].

Therefore, in theory, the exact frequent subgraph mining problem becomes computationally intractable for large graphs or low thresholds. Furthermore, it is also practically intractable for the majority of real-world graph databases [265]. This is evident from the fact that already the problem of counting a specific subgraph in a graph database is $\mathcal{NP}$-hard [107, 151, 265]. Only for very specific subgraph types, such as graphs with bounded treewidth, there are polynomial delay[17] algorithms available [340, 342].

An alternative to exact frequent subgraph mining is *approximate frequent subgraph mining* [341], where the goal is to only find a (random) subset of all frequent subgraphs.

Besides frequent subgraph mining, there are other graph pattern mining tasks, most notably compression-based subgraph mining [170] such as Subdue. Subdue has also been one of our main inspirations for the compression-based approach we will employ in Chapter 6. In compression-based subgraph mining, the goal is to find the most compact representation of the graph database that still allows for the reconstruction of the original graphs. The idea is to find a set of subgraphs that can be used to compress the graph database, and to use these subgraphs as a summary of the database (cf. Section 3.2.2). Recently, Bariatti et al. [28] proposed a graph mining approach based on the Minimum Description Length principle, which is also a compression-based approach.

3.3.2 Graph Representation Learning and Graph Neural Networks

As outlined in Section 3.2, we are interested in the identification of reoccurring patterns in graphs. When it comes to pattern mining in graph data, a reoccurring problem is to identify whether two graph representations refer to the same graph. Although the problem of determining whether two given graphs are identical ap-

17 That is, the time between the output of two subgraphs is polynomial in the size of the input database.

pears to be a straightforward one, no polynomial-time algorithm is currently known to exist for this task [24]. In general, for many graph mining tasks it is crucial to know how similar two given graphs are. For example, in graph clustering, a similarity measure[18] or a distance measure is needed to group similar graphs together. In fact, many applications in the area of artificial intelligence and machine learning have been proposed for data in the Euclidean space, where the distance between two data points is a measure of their similarity. Since Euclidean space and Euclidean distance are not suitable for all types of data, researchers have developed kernel methods. These involve mapping arbitrary spaces into a Hilbert space where computations such as computing an inner product—and thereby deriving distance measures—can proceed in a structured manner [287].

In particular, approaches for the *representation of graphs* and their comparison have been proposed [121]. The idea of a representation is to map a graph to a Hilbert space[19], for example, $\mathbb{R}^n$ equipped the standard (Euclidean) scalar product

$$\langle x, y \rangle = \sum_{i=1}^{n} x_i y_i,$$

such that the distance between two representations x and y, that is,

$$d(x, y) = \sqrt{\langle x, x \rangle - 2\langle x, y \rangle + \langle y, y \rangle}$$

is a measure of the distance between the two graphs.

Example Adjacency matrix and list as graph representation.
The most straightforward approach is to use the adjacency matrix[a] (or likewise similar matrix representations for labeled graphs) of a graph as a representation. However, for sparse graphs, this representation is not very efficient, and for one graph, there are many equivalent adjacency matrices. Even though these adjacency matrices are equivalent in the space of graphs, considered as a vector in $\mathbb{R}^{n^2}$—equipped with the Euclidean distance measure, or an angle-based similarity measure—they are different. The same holds true for more space-efficient representations such as the edge list or the adjacency list.

a The adjacency matrix of a graph with n nodes is an $n \times n$ matrix a_{ij} with $a_{ij} = 1$, if there is an edge in the graph between node i and node j, and $a_{ij} = 0$ otherwise.

18 Note that, given a distance measure d, a similarity measure can be defined as $s(x, y) = e^{-\alpha d(x,y)}$. Also, other strictly monotonic transformations are conceivable. All of them have in common that the notion of "closer" is independent of whether one considers a distance, or a similarity measure. That is, similarity and distance are dual concepts.

19 A Hilbert space is a generalization of Euclidean space. That is, it has a vector space structure (also called linear space) and an inner product that makes it complete (i.e., every Cauchy sequence converges w.r.t. to the distance induced by the inner product.).

Graph Kernels

A more promising, more general approach is to introduce a (graph) kernel function [110, 124, 334]:

Definition 3.3.3 Kernel Function.
Let $\mathcal{G}$ be a set of graphs. A kernel function *is a function $k : \mathcal{G} \times \mathcal{G} \to \mathbb{R}$ with two properties:*

1. Symmetry*: $k(g, g') = k(g', g)$ for all $g, g' \in \mathcal{G}$, and*
2. Positive-definiteness*: For all $n \in \mathbb{N}$, all $g_1, \ldots, g_n \in \mathcal{G}$, and all $\alpha_1, \ldots, \alpha_n \in \mathbb{R}$, the inequality $\sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j k(g_i, g_j) \geq 0$ holds.*

A kernel function k is called complete*, if the mapping $g \mapsto k(g, \cdot)$ is injective, that is, if $k(g, \cdot) = k(g', \cdot)$ implies g and g' are isomorphic graphs.*

The kernel function k can be seen as a generalization of the inner product, and therefore naturally leads to distance measures and similarity measures. Indeed, the relationship to Hilbert spaces is more striking, as the following standard result—Mercer's Theorem—in functional analysis shows [124]:

Theorem 3.3.2 Mercer's Theorem.
Let $\mathcal{G}$ be a set of graphs. Let k be a kernel function. Then there exists a Hilbert space $\mathbb{H}$[a] and a mapping

$$\phi : \mathcal{G} \to \mathbb{H}, \text{ such that}$$
$$k(g, g') = \langle \phi(g), \phi(g') \rangle_{\mathbb{H}}.$$

a Typically, $\mathcal{H}$ is used to denote a Hilbert space. Instead, to avoid confusion with a set of hypothesis, also denoted by $\mathcal{H}$, we use the symbol $\mathbb{H}$ to denote a Hilbert space.

In practice, $\mathbb{H}$ is often chosen to be the space of square-summable sequences l_2 and ϕ becomes a set of feature maps $(\phi_i)_1^n$ where each ϕ_i is a function on the space of graphs $\mathcal{G}$. For complete kernel functions, we get an interesting result to compare graphs:

Theorem 3.3.3
Let $\mathcal{G}$ be a set of graphs. Let k be a complete kernel function. Then, for non-isomorphic graphs g, g' in $\mathcal{G}$, there is at least one i such that $\phi_i(g) \neq \phi_i(g')$.

Proof. Per definition, since k is complete, the mapping $g \mapsto k(g, \cdot)$ is injective. Then the mapping $g \mapsto k(g, \cdot)$ is injective. Therefore, at least one of the feature maps ϕ_i

is different for non-isomorphic graphs g and g', because otherwise $k(g,\cdot) = k(g',\cdot)$, which would imply g and g' are isomorphic. □

Therefore, for a complete kernel function k, the feature maps ϕ_i are *graph invariants*[20] and the set $(\phi_i)_1^n$ is a *complete set of graph invariants* that uniquely identifies a graph. This ultimately yields the following rather disillusioning result (see also Gärtner [110, 263]):

Theorem 3.3.4
Given g, g' in $\mathcal{G}$, the problem of computing a complete graph kernel $k(g,g')$ is at least as hard as determining whether g and g' are isomorphic.

Since computing graph isomorphism is thought to be not polynomial-time computable, the same holds true for computing a complete graph kernel.

Example 3.3.3 Subgraph Isomorphism Kernel.
One can design a kernel function via its feature map for the subgraph isomorphism problem. For every graph $h \in \mathcal{G}$, we define the feature map ϕ_h to count the number of occurrences of h in a graph g. The kernel function is then defined as $k(g,g') = \sum_{h\in\mathcal{G}} \phi_h(g)\phi_h(g')$. This kernel function is complete, as it is injective, and the feature maps are graph invariants. Unfortunately, as we know from theorem 3.3.1, the problem underlying the computation of this kernel function is $\mathcal{NP}$-hard.

In practice, we have to trade off the expressiveness of the kernel function against the computational efficiency of computing it. [263]

Therefore, several approximations have been proposed to compute graph kernels, such as the random walk kernel (k counts the number of identical random walks on g and g'), subgraph kernels (similar to Example 3.3.3, k counts the number of common subgraphs of g and g', but for a limited, pre-defined set of small subgraphs), graph kernels based on subtree patterns (similar to random walk kernels, but the random walks are replaced by subtree patterns), and graphs kernels based on the Weisfeiler-Lehman algorithm (a graph isomorphism test based on the Weisfeiler-Lehman algorithm) [299]. The Weisfeiler-Lehman algorithm allows for a fast computation of a property of a graph that is invariant under isomorphism. It can not be used to decide if two graphs are isomorphic, but when this property is different for two graphs, they are not isomorphic. As the Weisfeiler-Lehman algorithm is also pertinent to other graph representations, we will undertake a more detailed examination of this example.

20 A graph invariant is a function that assigns a scalar value to a graph, such that isomorphic graphs have the same value.

Example Weisfeiler-Lehman graph kernel [299].
The Weisfeiler-Lehman graph kernel is based on the Weisfeiler-Lehman algorithm, which is a graph isomorphism test. The algorithm assigns a label to each node in the graph, based on the labels of the nodes in the neighborhood of the node. To be more precise, the algorithm assigns a label to each node in the graph, based on the sorted[a] multiset (i.e., repetitions in the set are allowed) of labels of the nodes in the node's neighborhood. For example, when a node with label a has two neighbors with labels b and c, the node will be labeled abc—and, in practice, this label will often be compressed.
The algorithm is then iterated using the graph with augmented labels as input. One can show that this iteration will eventually converge to a fixed point, where the labels of the augmented nodes do not change anymore—which takes at most n (number of nodes) iterations.
By this iteration, one obtains a sequence of graphs, $(g_0, \ldots, g_h)$, where g_0 is the original graph, and g_h is the graph computed by the Weisfeiler-Lehman algorithm with h iterations.
Given a base kernel k, The Weisfeiler-Lehman graph kernel is then defined on this sequence as

$$k_{KL}^{(h)}(g, g') = \sum_{i \leq h} k(g_i, g_i')$$

It can be shown that $k_{KL}^{(h)}$ is positive-definite for a positive-definitive base kernel k. [299]
A straightforward application would be a base kernel that counts the number of common labels for a graph. It can be shown for the resulting kernel $k_{KLsubtree}^{(h)}$ that it has a complexity of $\mathcal{O}(hm)$, where h, as before, is the number of Weisfeiler-Lehman algorithm iterations and m is the number of edges of the graphs. [299] In experiments, the Weisfeiler-Lehman graph kernel above has been shown to be a good compromise between expressiveness and computational efficiency. [299]

a Sorting is not really required for the algorithm, because any injective function from the set of multisets to the set of natural numbers can be used.

Graph Neural Networks

Kernels and feature maps for graphs are a powerful tool in many machine learning and mining applications for graphs. [350] For example, kernels can be used in support vector machines, or in other kernel methods such as kernel PCA, kernel k-means, or kernel regression. However, the computation of kernels can be computationally

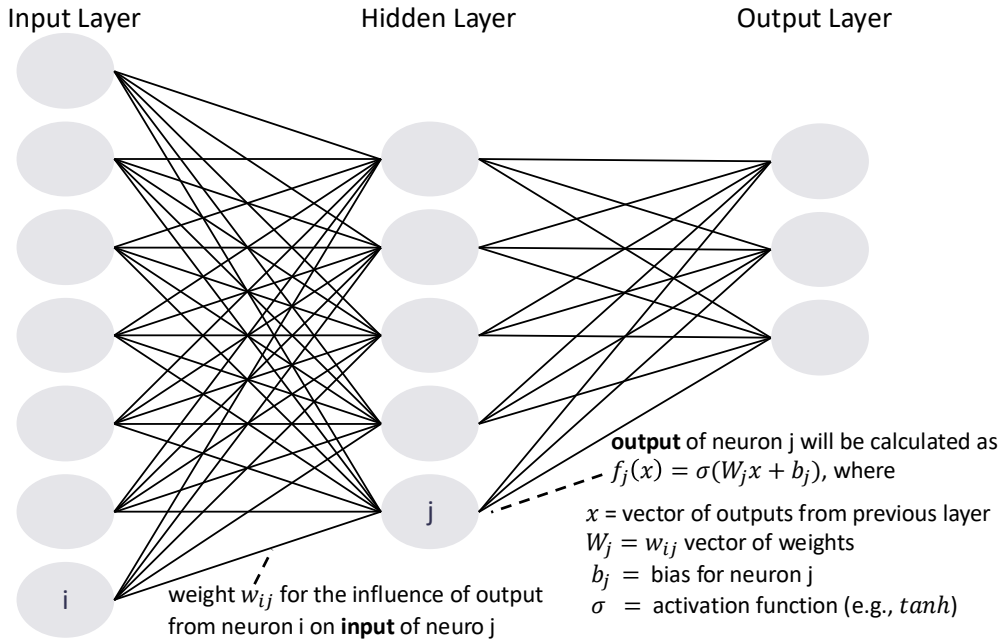

Figure 3.5: A depiction of a neural network. For a given input, every neuron will have a real-valued output, forwarded to the next layer or given as output, if the neuron is in the output (i.e., the last) layer. The concrete example would compute a function $f\colon \mathbb{R}^7 \to \mathbb{R}^3$.

expensive, and the choice of a (good) kernel function in a specific context is often not straightforward.

Therefore, researchers have proposed to learn the feature maps directly from the data, using neural networks. That is, for a specific use case, for example, graph classification, one can use machine learning on the graphs, to implicitly learn a representation that is optimal for the task at hand. For this thesis, we are not interested in machine learning on graphs in general, but only in generative approaches which we will introduce in Section 3.4. In order to understand the ideas behind these generative approaches, we will briefly introduce the concept of graph neural networks (GNNs) [350, 353]. We therefore have to introduce the concept of machine learning, neural networks, and then graph neural networks. The basic idea of machine learning, in general, is to learn a function f based on given data. For example, in the case of supervised machine learning, f is learned from pairs of input and output data (x, y), where $f(x) \approx y$. Neural network can be seen as a concrete realization of this idea, where the function f is approximated by a parametric function f_θ, where θ are the parameters (i.e., weights and biases) of the neural network. An example is given in Figure 3.5.

Definition 3.3.4 Neural Network, Graph Neural Network.
A neural network *is a function* $f : \mathbb{R}^n \to \mathbb{R}^m$ *that is composed of several* layers *of functions. Typically, every layer consists of linear functions, followed by a non-linear* activation function, *such as the sigmoid function, the hyperbolic tangent function, or the rectified linear unit (ReLU) function. One specific neuron* i *gets its input from all neurons in the previous layer, and computes the output as*

$$f_i(x) = \sigma(W_i x + b_i), \tag{3.7}$$

where W_i *is the* weight matrix, b_i *is the* bias vector, *and* σ *is the* activation function *for neuron* i.
A graph neural network *is a neural network that is designed to operate on graph-structured data. Similar as in the Weisfeiler-Lehman algorithm, nodes in the graphs are updated based on the labels of the nodes in the neighborhood. This is called* message passing, *and is a generalization of Graph Convolution Networks [176]. Let* v *be a node in a graph, and* $\mathcal{N}$ *be the set of neighbors of* v, *and* $h_v^{(k)}$ *be the feature vector (representation) of node* v *in layer* k. *The output feature vector of the neural network is then iteratively computed by a neural network through the following steps:*

1. Aggregation*: The feature vectors of the neighbors are aggregated to a single feature vector for every layer* $k > 1$ *of the neural network.*

$$a_v^{(k)} = \text{aggregate}^{(k)}\left(\left\{h_u^{(k-1)} \mid u \in \mathcal{N}\right\}\right)$$

2. Combination*: The nodes aggregated "messages"* $a_v^{(k)}$ *are combined with the feature vector of the node* v *from the previous layer.*

$$h_v^{(k)} = \text{combine}^{(k)}\left(h_v^{(k-1)}, a_v^{(k)}\right)$$

Typically the input $h_0^{(k)}$ *is given by node label representation. It is also possible, to extend this scheme for edge labels, for example, aggregating node features and edge features along the message passing. For graph level tasks, one typically defines a further aggregation on all nodes to obtain a graph-level representation, that is, the representation of the entire graph* g *is then given by*

$$h_g = \text{readout}\left(\left\{h_v^{(K)} \mid v \in \mathcal{V}\right\}\right),$$

where K *is the number of layers in the neural network.*

Choosing an aggregation scheme "aggregate", a combination scheme "combine", and a readout scheme "readout", one can design different graph neural network

architectures. Xu et al. [353] have shown that graph neural networks can be at least as powerful as the Weisfeiler-Lehman algorithm. One requirement, in order to not map different node environments on the same representation, is that the aggregation scheme "aggregate" is injective. Furthermore, the readout scheme "readout" should be permutation invariant (e.g., a summation, but not a concatenation of the node representations), that is, the order of the nodes in the graph should not matter.

The following theorem shows the expressive power of neural networks:

Theorem 3.3.5 Universal Approximation Theorem [135].
A neural network with one hidden layer and a non-linear activation function can approximate any continuous function on a compact subset of $\mathbb{R}^n$ *arbitrarily well.*

Furthermore, neural networks can be efficiently trained using automatic differentiation and optimization algorithms such as stochastic gradient descent (SGD) or its variants, such as Adam [173]. Of course, this does not mean that we can apply any arbitrary neural network at a problem, and it will solve it. The architecture of the neural network, the choice of the activation function, the initialization of the weights, weight normalization, and the optimization algorithm are crucial for the success of the learning process, and have to be chosen for the specific problem at hand.

Example
Graph neural networks have been used to design approximate solutions the subgraph isomorphism problem (see Section 3.3). For example, Chen et al. [63] have investigated several graph neural network architectures for subgraph isomorphism counting. In their experiments they have trained several architectures to count five fixed subgraphs (with 3 and 4 nodes) in a graph. For these (small) subgraphs, they have shown that the graph neural networks can approximate the number of occurrences of the subgraphs in a graph to a large extent.

Remark 3.3.1
Representations learned by neural networks can be used as the feature maps in a kernel function, and thus can be used to compute a graph kernel. However, the computation of the kernel function is not necessary, as the neural network can be used directly for the task at hand. Graph neural networks (which to some extent are certainly motivated by graph kernels) and graph kernels share many commonalities as they both aim to bridge the gap between graph-structured data and distributed representations (in Euclidean space), for which a distance measure can be defined. Both methods are designed for graph-structured data, by ensuring that the representations are invariant under isomorphism. However, graph neural networks are more flexible, as they can be

trained end-to-end for a specific task, based on a dataset, while graph kernels are fixed once they are defined.

3.4 Generative Models

In example 3.2.3, we described the evolution of software models by the assembly via edit operations (cf. Section 2.4). Given a current state of a software model, m_0, a "human designer" chooses to apply an edit operation ε to m_0 to obtain a new model, m_1, via the assembly step (also called edit operation application) $m_0 \xrightarrow{\varepsilon} m_1$.

Forgetting about the human designer involved in it, this design process will actually resemble a generative process, where a *generative model* continually generates a sequence of (valid) software models. Measuring this generative process, we would actually find, that the process is not purely random, but at least conditioned on the current state of the model.

Definition 3.4.1 Generative Model.
A stochastic process $\{X_t\}_{t\in\mathbb{N}}$ generating a sequence of random variables $X_t \in \Omega$ according to a probability distribution

$$\mathbb{P}(X_{t+1}|X_t, X_{t-1}, \ldots, X_1), \tag{3.8}$$

is called a generative model.

Example 3.4.1 Software Modeling as a Generative Model.
For the example of the software model assembly, we would have $\{m_t\}_{t\in\mathbb{N}}$ as the sequence of software models, where each $m_t \in \mathcal{M}$ is a software model. In case m_{t+1} only depends on m_t, that is, the choice of the edit operation ε applied to m_t is independent of the history $m_0 \to \cdots \to m_t$, the generative model becomes a Markov process.

At first glance, this model might seem to be a very narrow concept. However, consider the example of a deterministic $\mathbb{P}$ that, given some initial state X_t, computes a function $f : \Omega \to \Omega$, where Ω can be an arbitrary probability space (e.g., $\Omega = \mathbb{R}$). This shows, that the above definition is rather broad. In particular, it is capable of modeling a design process, provided that all the requisite knowledge is encoded in the modeling history and the probability distribution $\mathbb{P}$.

In artificial intelligence and statistics, many methodologies have been developed to model generative processes [123]. Most recently, very powerful generative neural network architectures have been developed, such as *Generative Adversarial Networks*

(GANs) [115], *Variational Autoencoders* (VAEs) [174], *Diffusion Networks* [306], and *Generative Graph Neural Networks* (GGNNs) [206].

3.4.1 Generative Graph Neural Networks

In the context of the evolution of software models, a natural choice for a generative model would be one that can handle graph-like data. Several architectures for generative graph neural networks have been proposed in the literature [175, 206]. For this work, we will choose the *Graph Variational Autoencoder* [175] as a generative model. The idea of an autoencoder, in general, is to first encode the input data into a *latent space* with an *encoder* model and then use a *decoder* model to decode the latent space representation back into the original input data. A reconstruction loss can be used to train the model to minimize the difference between the input data and the decoded data. Therefore, an autoencoder can be used to learn representations of the input data (cf. Section 3.3.2) in an unsupervised manner (i.e., without target variables). A problem with standard autoencoders is that using only the decoder part as a generative model, parts of the latent space would not be meaningful, and the generated data might not resemble the input data well [103]. This is, because in the distribution of the training data in the latent space can be very heterogeneous, with a large spread and gaps where no data points are present. Especially, when sampling from these gaps, there is no reason to assume that the generated data will resemble the input data. A solution to this problem is to introduce a probabilistic model in the latent space, "forcing" the original data to follow a certain distribution in the latent space. This is the idea behind the Variational Autoencoder [174].

Definition 3.4.2 Graph Variational Autoencoder [174, 175].
A Variational Autoencoder *is a generative model that extends the standard autoencoder by introducing a probabilistic model in the latent space. The Variational Autoencoder is trained to maximize the likelihood of the input data under the generative model. A* Graph Variational Autoencoder *is a variational autoencoder that is specifically designed to handle graph-like data. Given a representation x of a graph $g \in \mathcal{G}$ (e.g., an adjacency matrix representation), there are two probability distributions, the* encoder $q_\phi(z \mid x)$ *and the* decoder $p_\theta(x \mid z)$*, where z is the latent space representation. The network is then trained to maximize the likelihood of the input data under the generative model.*

$$L(\theta, \phi) = \underbrace{-E_{z \sim q_\phi(z|x)} \left[\log p_\theta \left(x \mid z\right)\right]}_{\textit{Reconstruction Error}} + \underbrace{D_{KL} \left(q_\phi \left(z \mid x\right) || \, p(z)\right)}_{\textit{KL Divergence}} \tag{3.9}$$

p, and q will be parametrized by neural networks with weights and biases θ and ϕ. The prior $p(z)$ is typically chosen to be a standard normal distribution, that is, $p(z) = \mathcal{N}(0, I)$.

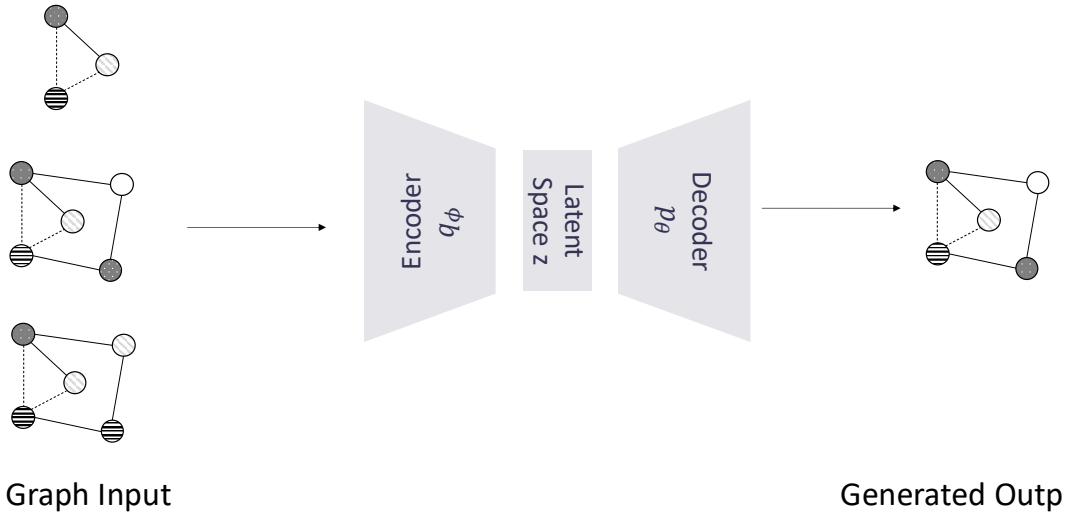

Figure 3.6: A depiction of a Graph Variational Autoencoder. Input graphs are encoded by a parametrized encoding distribution q_ϕ and decoded by a parametrized decoding distribution p_θ. Sampling from the latent space z and using the decoder after training gives a generative model for graphs.

A depiction of a Graph Variational Autoencoder is shown in Figure 3.6. The encoder $q_\phi(z \mid x)$ will be a Graph Neural Network that encodes the graph x into the latent space z. The decoder is often chosen to be a simple neural network that decodes the latent space representation back into the graph representation. As can be seen from Equation 3.9, the loss function of a Graph Variational Autoencoder consists of two terms: the *Reconstruction Error* and the *KL Divergence*. The *Reconstruction Error* term is the negative log-likelihood of the input data under the generative model—it is a measure of how well the model can reconstruct the input data. The *KL Divergence* term in the loss function can be seen as a distance between the prior distribution and the encoder distribution in the latent space. This term in the loss function therefore ensures that the latent space representation is close to a standard normal distribution and therefore avoids the problem of non-variational autoencoders that parts of the latent space might not be meaningful. Sampling normally distributed from the latent space and decoding the samples with the decoder part of the network will generate new graphs that resemble the input data—assuming the network has been trained properly. That is, we obtain a generative model for graphs $\mathbb{P}(g)$ for $g \in \mathcal{G}$. This generative model is unconditioned.[21]

21 For this thesis, regarding graph neural networks, we will only investigate *unconditioned* generative models, although the concept can be extended to conditioned models as defined above.

As we are also interested in pattern mining—in particular, the edit patterns in software models—one interesting question is if patterns existing in the input data are stored in the neural network parameters. As we have discussed in Section 3.2.2, patterns naturally arise in the generation of data via a program. The patterns serve as a kind model for explaining data. In Section 3.2.2, we were mainly concerned with the lossless compression of data. Based on the same ideas of Algorithmic Information Theory and Minimum Description Length Principle, one can also investigate lossy compressions of data. Both, Algorithmic Information Theory and the Minimum Description Length Principle, are also concerned with lossy compression, with the idea that noise or irrelevant information in the data can be discarded. Roughly speaking, the idea is, the less some information can be compressed the less meaningful it is.

As suggested in Figure 3.6, information from the input to the encoder passes through a (variational) autoencoder and will finally be compressed into the—typically lower-dimensional—latent space. If the compressed information in the latent space is sufficient to reconstruct the input data, the information in the latent space can be seen as meaningful. It is therefore an interesting question to ask whether the patterns in the input data are stored in the latent space representation. We later investigate this question empirically in Chapter 7.

3.4.2 Language Models

An interesting class of generative models $\mathbb{P}(X_{t+1}|X_t, X_{t-1}, \ldots, X_1)$, is a stochastic process X_t of words, or tokens[22], that is, a sequence of words or tokens. This class of generative models, for obvious reasons, is referred to as *Language Models*. Language, for sure, is subject to evolution [100]. As for software models, we therefore expect observed language data to be *biased*—in particular, not every sequence of words is equally likely to occur in a natural language text. A good language model should be able to capture this bias and generate text that resembles natural language.

Typically, we denote the language model stochastic process by $\omega_t := X_t$. That is, ω_t is the token generated at time t. Furthermore, we define the *context* c by $c := \omega_1 \ldots \omega_t$ (i.e., concatenated tokens already generated by a process). We then have the following definition for a language model:

Definition 3.4.3 Language Model.
A language model *is a conditional probability distribution* $\mathbb{P}(\omega_{t+1}|c)$ *for a* token ω_{t+1}*, given a* context c*. When the context is a predefined (i.e., not generated) sequence of tokens, we also refer to* c *as a* prompt.

22 A token is a sequence of characters, usually more than one character but less than a word. Tokens have been proven to be successful in Natural Language Processing. Tokens are learned from large text corpora and there is no simple rule to define what constitutes a token. A *tokenizer* can map text to tokens and vice versa.

Historically, linguists have been interested in models for natural language for a long time. In the 50s, Noam Chomsky introduced the idea of *generative grammars* [65], which are formal systems that generate sentences of a language. Statistical methods have also been investigated in the past, for example, Jelinek and Mercer [148] derive the probabilities of so-called *n-grams* (i.e., sequences of n tokens) from a *corpus* (i.e., a set of documents). A problem with n-gram models is that one can construct n-grams that are syntactically correct and meaningful but never occur in the corpus from which the n-gram model is derived. It is not clear which probabilities to assign to these "unseen" n-gram. In the past, attempts have been made to reduce the context size, that is, fall back to the smallest context for which a prediction could be made. Furthermore, interpolation techniques have been used [148].

Probably, these difficulties in the application of statistical models for the generation of natural language have been one reason why grammar-based approaches have been more popular until the early 2000s. In particular, it is worth mentioning that the basic ideas of today's language models have been around for a long time. Already in 1954, Zellig S. Harris hypothesized that the semantics of a word is determined by the words that frequently occur in its proximity [122]. Later, in the 80s, motivated by the representation of concepts and language in the human brain, the idea of *distributed representations* [133] has been developed. The idea of distributed representations is that a word should not be represented by an index in a dictionary, but rather by a vector in a high-dimensional space. Efficient representations of (sequences of) words avoid handling sparse n-gram tables and can carry linguistic and semantic information. Similar to graph representations discussed above, distributed representations can leverage Hilbert space structures to make the meaning of *similarity* mathematically precise. Fusing the ideas of distributed representations and distributed semantics, Bengio et al. [32] proposed using neural networks to learn the probability distribution $\mathbb{P}(\omega|c)$. Based on this idea, Bengio et al. [32] proposed using neural network architectures to learn the probability distribution $\mathbb{P}(\omega|c)$. In 2013, Mikolov et al. [228] introduced the *Word2Vec* model, which learns distributed representations—so-called *word embeddings*—of words by predicting a word from its context. With the success of transformer architecture [330], language models have become quite popular and are used in plenty of domains including software engineering [62, 279, 320, 351, 361].

Today neural language models with billions of parameters are *pre-trained* on large corpora of natural language text as well as source code. These pre-trained models can then be *fine-tuned* for specific data sets and applications. Especially remarkable was the observation that pre-trained large language models are multitask learners, that is, they can be used for a variety of tasks such as text classification, question answering, language translation—without training on large data sets [262]. In particular, even without fine-tuning, only using the context of the model, many tasks can be solved by a pre-trained large language model.

Remark 3.4.1
A key difference of the language models introduced in this section, and the generative models introduced in Section 3.4.1 is that language models are naturally compatible to the idea of evolution, *because they basically answer the question: "What is the next word in a sequence of words?" The models we introduced in Section 3.4.1 can also be* extended *to a similar question, that is, "What is the next node or edge in a graph?" [87, 339, 357]. The major reason why we investigate* unconditioned *graph generative models that are not directly trained on time series of graphs is about the maturity of the field. That is, for graph generative models, we will rather focus on whether graph patterns are represented in the latent space, and less on their predictive power for evolution. Large language models, on the other hand, are a mature technology and have shown promising capabilities in code generation and program synthesis [62, 279] and their investigation in model-based software engineering is therefore conceivable.*

Even though large language models have been developed for natural language, they have also shown preliminary promising results for tasks on graph-like data [64]. In fact, in many use cases, sequential (textual) data describes a more complex structure—for example, source code is a representation of an executable program. For graph like data, a common approach is to serialize the graph and combine it with some context and the task description. A generated prompt will then be the input to a large language model (see Figure 3.7). Alternatively, a language model can be fine-tuned on a specific task described by a dataset of serialized graphs.[23]

Example Graph Generation with Large Language Models.
To generate graphs that resemble the graphs in a graph database with large language models, we can sample graphs from the input database (how many depends on the context size the language model is capable of handling) and concatenate the serialized graphs a task description:

```
The following text represent serialized graphs. Nodes are
    represented [description of the graph format]. Please
    generate new graphs that resemble the input graphs and share
     their graph properties (e.g., label distribution, node
    degree distribution). Please do not generate duplicates,
    that is graphs that are isomorphic to one of the given
    graphs. Generate novel graphs that are as close as possible
    to the input graphs:

t # g1
e 0 1 solid dotted diagonal
```

23 Combinations of graph neural networks and language models have also been proposed [64] and show promising results for many tasks. Anyway, in this thesis we will not further look into this direction.

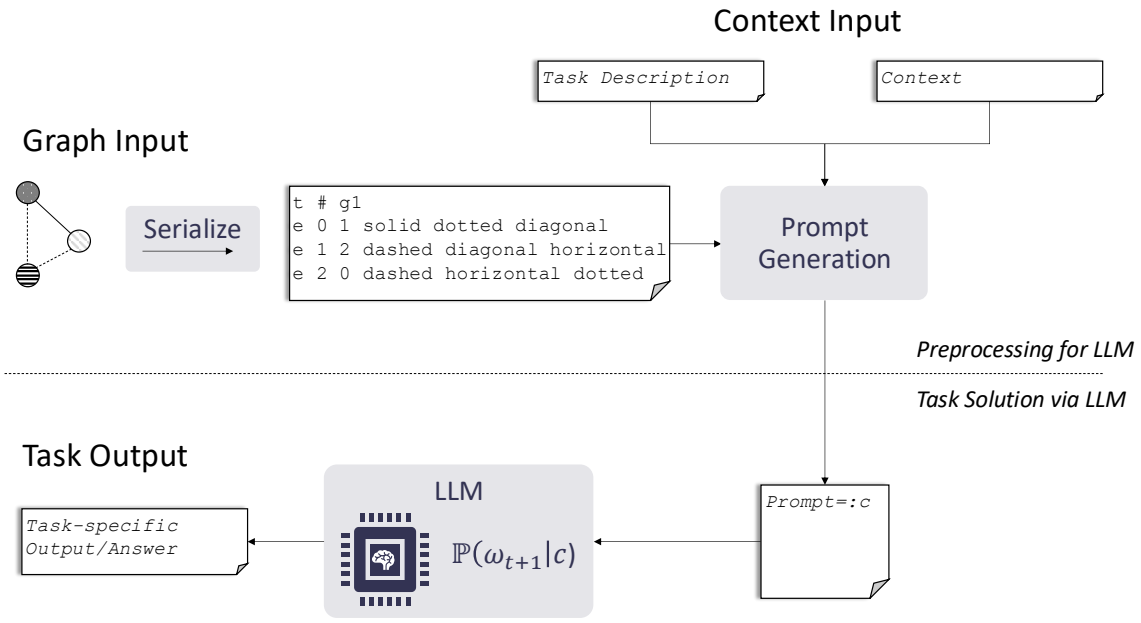

Figure 3.7: A depiction of a general blueprint to solve tasks on graphs via large language models, which have been designed for language input.

```
e 1 2 dashed diagonal horizontal
e 2 0 dashed horizontal dotted

[...]

Novel graphs:
```

Indeed, in a pilot study we compared the performance for graph generation of a Graph Attention Network (GAT) GNN with that of GPT-4 on a graph database in Cheminformatics (the ZINC dataset) and a dataset of serialized Ecore models. In this pilot study, we found that for many graph properties, GPT-4 outperformed the GAT GNN. In particular, we found that the GNN was similar or superior to GPT-4 for graph properties that were explicitly encoded in the GNN architecture (e.g., the node label distribution). For graph properties that were not explicitly encoded in the GNN loss function (e.g., node centrality), GPT-4 clearly outperformed the GNN.

We will investigate an approach to model auto-completion based on these ideas in Chapter 8.

Remark 3.4.2
A degree of freedom that comes with this approach is in the choice of the serialization. There is not only a plethora of different serialization formats for graphs, but also there are many orders of nodes and edges for every single graph. For example a connected subgraph can be disconnected in a serialization and vice versa. That is even if nodes that are "semantically related" in the original graph, they might be disconnected in the serialization and therefore Harris' distributional hypothesis might not hold. The smaller the relative size of a pattern compared to the graph it is embedded in, the higher the probability that the pattern is disconnected in the serialization. As long as no further rules are applied to counter this effect, one can only hope that in statistical sense, a language model captures the patterns in the input data.

3.5 A Theory of Software (Model) Evolution from a Graph Mining and Generative Model Perspective

The answers you get depend on the questions you ask.
— Thomas Kuhn

We have connected Algorithmic Information Theory and complexity measures, in general, to patters in software models in Section 3.2.2. In Section 3.3, for graph-like data such as abstract syntax graphs (see Definition 2.3.1) or simple change graphs (see Definition 2.4.4), we have seen how techniques from the field of graph mining help extracting knowledge from the data, in particular, identifying frequent graph motifs. We have then made some remarks of the connection of evolution and generative models in Section 3.4, and have discussed two specific families of generative models: graph variational autoencoders and large language models.

In this section, we combine these ideas and show how they are related to the evolution of software models, and can be made practically useful—in particular, for the identification of patterns in software models and to complete software models.

3.5.1 An Economical Viewpoint of Evolution

Evolutionary economics postulates that technological evolution goes through a process of inventions, incremental innovation, and radical innovation [253] (see

Figure 3.8 for a depiction of this process).[24] *Inventions* are new ideas that are not yet implemented in a product. *Incremental innovations* are improvements of existing products, while *radical innovations* are new products that are fundamentally different from existing products. Not every invention will meet the market's needs. Only those that are *adopted* by the market will be successful.

An invention is typically assembled from parts by the help of tools or existing concepts. The more suitable the tools to build an invention for some market the easier the production of the product and therefore the lower the cost and consequently the more likely it is that inventions become innovations. An important question to optimally support the evolution of a certain market or ecosystem seems to be: What is the "best toolbox" to build the products (in a certain niche) that the market needs?

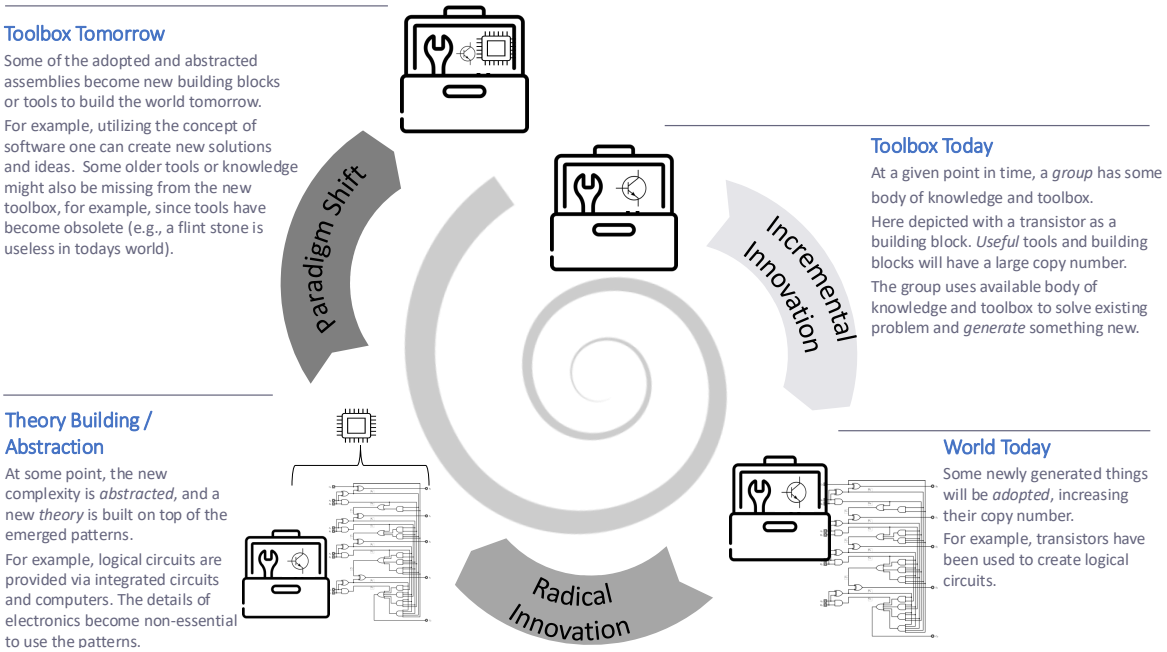

Figure 3.8: The duality of assemblies and tools: tools are used to build a "new thing". Then (parts of) the new thing become tools in the future. Theories and abstractions incorporating the new things and the concept of *information hiding* reduce the increasing complexity. We focus on the incremental innovation part and highlight that a similar process happens on small scale in software modeling.

Let us consider two specific examples to illustrate this argument:

Example House Construction.
How could we determine the optimal toolbox for house building? We could look at a collection of already built houses and determine which tools would

24 A similar process also happens for science and has been described by Thomas Kuhn [191] as the "Structure of Scientific Revolution".

have been most useful (in terms of costs, i.e., low material costs, labor costs, etc.) to build these houses. The costs of the tools (to buy them, learn to apply them, logistics to get them at the construction site, etc.) would constrain the set of tools. Having too many tools involved would definitely not be an ideal solution because it increases the difficulty in selecting the right tools during the construction process.

In the field of software modeling, we are facing a similar situation:

Example Software Modeling Tool.
Suppose a modeling tool offers a certain set of functionality. This functionality is provided to the user in the form of *edit operations*. Using the tools' functionality, new software models are created. Typically, usage patterns will arise (e.g., in a class diagram one might observe that users often pull-up attributes from two or more child classes to a parent class). To support the user in creating these models, the tool provider might decide to incorporate this pattern into the tool (e.g., by providing a pull-up refactoring edit operation). If this new operation is adopted by the tool users, it will typically stay in the tool and become a long-term functionality of the tool. Then other competing tools might incorporate this functionality as well. Even if the tool will be discontinued, useful functionality the tool has been providing can be incorporated into the next generation of tools. Similarly, in the other direction, if the tool developer suggest a new operation to the tool users the users basically decide if they will adopt this operation. If they adopt it, the application of this operation will lead to patters in the modeling history. In fact, it is conceivable that the user will only adopt a new edit operation provided by the tool, if it is useful for the task that the user needs to accomplish.

Every niche or market has its paradigms. For example, Model-based Software Engineering, has certain concepts, tools, and best practices how to use them. As part of the incremental innovations, a revolution cycle happens continuously.[25] New patterns will emerge and will be added to the "tool boxes", in the form of processes, best practices, laws, architectural blueprints, tools and edit operations, or even reusable building blocks or libraries. In this work, we will not study the process of radical innovation, that is, large scale revolutions (except for a short outlook in Section C). We also side-step questions of adoption and only emphasize here that socio-economic factors lead to the selection of certain patterns or even paradigms.

25 There is something which we typically do not observe on a small scale: Paradigms, after a (long) period of incremental innovation, face diminishing progress. Often this leads to a crisis, sometimes even with a fight between competing paradigms. This crisis will then be resolved by a paradigm shift.

On a very high level of abstraction, the evolution of a software model can be described as a map from a timescale to a state of the system:

$$\begin{aligned} \Phi : \mathbb{R} &\mapsto \mathcal{M} \\ t &\to \Phi(t), \end{aligned} \tag{3.10}$$

where $\mathcal{M}$ is the set of all possible valid models of the system, t is a point in time, and $\Phi(t)$ is the state of the system at time t.

To model the future state of the system, for the current point in time t_0, we would like to extend the function Φ from the past $t \in [0, t_0]$ to the future $t \in (t_0, \infty)$. In this section, we will link this idea to the concepts of graph mining, generative models discussed above.

3.5.2 Automatic Inference of Edit Operations

A difference between two model versions can be described as a (partially) ordered set of applications of edit operations, transforming one model version into the other. Comparing two models can thus be understood as determining the edit operation applications that transform one model into the other.

Since simple operations on the abstract syntax graph of a software model are very fine-grained and may lead to inconsistent models, and since structural model differences are too specific and only describe a very specific evolution step, there has been a long history of attempting to infer *relevant* edit operations (and model transformations in general) from available information (e.g., the meta-model, given examples, or a model history).

Definition 3.5.1 Automatic Inference of Edit Operations.
Given a meta-model $\mathcal{TM}$ with software model universe $\mathcal{M}$, automatic inference of edit operations *(i.e., edit operation mining) is a computable function $I\colon 2^{\mathcal{M}} \to 2^{\mathcal{E}}$ that, given a set of models (and their meta-model) $M \subset \mathcal{M}$, computes a set of edit operations $E \subset \mathcal{E}$.*

There are *supervised* and *unsupervised* approaches to the inference of edit operations. One branch of supervised approaches are demonstration approaches, were a tool user presents the transformation steps to an operation recorder [41, 359]. Typically, these approaches require some manual post-processing, for example, edit conditions have to be refined manually [41]. Another branch of supervised approaches include by-example approaches. Here, the tool user specifies a set of examples (and sometimes also counter examples) and the approach automatically infers the model transformations [15, 160]. These approaches have been motivated by the seminal work of Varró [329], who proposed by-example learning for exogenous model trans-

formations. Furthermore, heuristic approaches [234] have been proposed that apply search-based techniques to finding a set of refactorings.

Unsupervised approaches include generative approaches, deriving model transformations from the meta-model, and mining approaches. Generative approaches have been proposed in the area of model transformation testing [42]. Also, more recently, generative fuzzing approaches based on language models have been proposed that try to generate models with similar properties to real-world models [301]. Edit operations are only indirectly addressed within these generative approaches. It has been shown that a complete set of consistency preserving edit operations can also be derived from the meta-model [166, 218]. These operations capture static constraints that are already present in the meta-model and are typically very simple operations.

More recent model transformation by-example approaches also use neural networks to learn exogenous model transformations, for example, Burgueño et al. [47, 49] investigate learning exogenous model transformations from examples using long-short-term memory neural networks. However, these approaches need concrete input-output model pairs and have not been evaluated for endogenous model transformations.

A solution proposed in this thesis is mining model transformations from the modeling history using graph mining approaches. An advantage of mining approaches over by-example approaches is that they do not require handcrafting examples and therefore also fall in the category of unsupervised approaches. A disadvantage is their computational complexity and that negative application conditions and multi-object patterns (e.g., creating a variable count of elements in a model transformation) can not easily be inferred. Post-processing (e.g., using by-example approaches) of the mined operations is conceivable, though, and it this sense, by-example approaches and mining approaches are orthogonal.

3.5.3 Software Model Completion

Software model completion can be seen as an automatic model evolution on a very small timescale. In this subsection, we will formalize software model completion:

In software model completion, for an observed evolution $m \xrightarrow{\varepsilon} n$, we want to find a completion $\gamma \in \mathcal{E}$, such that $m \xrightarrow{\varepsilon} n \xrightarrow{\gamma} c$ is a meaningful (i.e., observable) completion. We call this γ a *completion operation*.

(Software) model completion is the task of further evolving a software model based on a given (partial) model. More formally:

Definition 3.5.2 Model Completion.
Given a set of model transformations $\mathcal{T}$, model completion *is a computable function*

$C\colon \mathcal{T} \to \mathcal{T}$ that, given a model transformation $m \xrightarrow{\varepsilon} n$ from a source model m to a (partial) target model n, computes a model transformation $C(m \xrightarrow{\varepsilon} n) = n \xrightarrow{\gamma} c$.

Example 3.5.1 Model Completion.
Suppose a port has been added to a component in a software model (conforming to the meta-model from Figure 2.2). A completion operation could then be the completion of adding a target port and connecting the ports via a connector. Given a meta-model for a class diagram, a completion operation could be the completion of adding a target class and connecting the class to the existing classes.

To implement model completion, some authors proposed to automatically complete a partial model to a model that conforms to the meta-model and possibly other constraints (e.g., via rule based approaches or constraint solving [244, 292, 309, 318]). However, these approaches are only able to complete a partial model to a model that conforms to the meta-model or satisfies additional explicitly given constraints. Other works take existing model repositories or pattern databases into account and employ model clone detection to discover similar (parts of) existing models to make completion recommendations [66, 311]. Fine-tuned large language models would come in here very handy, because they encode the software model history in their parameters and can provide context-specific completions. They would not require hand-crafted pattern databases or expensive clone detection in the model repository and are able to generalize to unseen context information. An overview of model completion approaches is given in the secondary study by Almonte et al. [14].

3.5.4 Generative Models for Software Model Evolution

We have seen that the modeling process can be described as a generative model as in Example 3.4.1:

$$\mathbb{P}(m_{t+1}|m_t, m_{t-1}, \ldots, m_0) \tag{3.11}$$

This model is a probabilistic and discrete time formulation of the more general model $\Phi(t)$ from Equation 3.10 that we have introduced at the beginning of this section. A discrete time formulation is sufficient for our purposes, as the most atomic operations are user clicks or keystrokes, which are discrete in nature.

In this section, we will dive into the details of how to construct such a generative model, understand the relationship between these generative models and edit operations, understand the connection to large language models, and finally consider their application in the use case of software model completion.

The simplest model for $\mathbb{P}(m_{t+1}|m_t, m_{t-1}, \ldots, m_0)$ would be to define a uniform distribution on a set of atomic edit operation (e.g., a set of consistency-preserving edit operations derived from a meta-model [166]) to obtain m_{t+1} from m_t. It is straightforward to see via combinatorics that the space of all possible models (for a given meta-model) is huge, and only a little fraction of models is conceivable. A random model like this would therefore be doomed to be useless—not even considering that the assumption on an i.i.d. process $\{m_t\}_{t\in\mathbb{N}}$ with uniform distribution on the set of edit operations is not realistic. Deriving $\mathbb{P}(m_{t+1}|m_t, m_{t-1}, \ldots, m_0)$ from a data set of observed models $M \subset \mathcal{M}$ seems more promising.

Instead of working directly on models, for the ease of notation, we can move to the space of model transformations (i.e., pairs of models) and rewrite 3.11 as

$$\mathbb{P}((m_{t+1}, m_t)|(m_t, m_{t-1}), \ldots, (m_1, m_0)). \tag{3.12}$$

For simplicity, let us assume for now, that we want to derive an unconditioned generative model for model transformations, that is, we want to derive $\mathbb{P}(m_{t+1}, m_t)$.

We could then define a probability measure on the set of all possible model transformations:

$$\mathbb{P}((m_{t+1}, m_t)) := \frac{\text{supp}_D(SCG((m_{t+1}, m_t)))\,\text{size}(SCG((m_{t+1}, m_t)))}{\sum_{g\in D}\text{size}\,(g)}, \tag{3.13}$$

where size denotes a measure for the size of a graph, for example, the number of the nodes plus the number of edges, and D is the space of all simple change graphs of all observed model transformations. This probability measure also has its pitfalls: With increasing size of $SCG((m_{t+1}, m_t))$ it becomes unlikely that exactly the same model transformation has already been observed in the model history.

We therefore need to find some other way to derive $\mathbb{P}((m_{t+1}, m_t))$ from M. As we have discussed in Section 3.2.2 the modeling process can also be seen as a sequence of edit operations that are applied to a model giving us a more detailed view on how $\mathbb{P}(m_{t+1}|m_t, m_{t-1}, \ldots, m_0)$ can be computed. More concretely, we assume a set of edit operations $E \subset \mathcal{E}$ and a mechanism (as defined by Ehrig et al. [90]) that applies these edit operations to a model. Based on the idea of edit scripts, that is, sequences of edit operations, we define the *edit history* of a model and the *shortest edit histories*.

Definition 3.5.3 Edit History.
Given a model $m \in \mathcal{M}$ and a set of edit operations E, an edit history *p_m of m is the sequence of edit operations $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n$ that transforms the empty model $\perp$ to m, that is,*

$$p_m := \perp \overset{\varepsilon_1}{\to} m_1 \overset{\varepsilon_2}{\to} m_2 \overset{\varepsilon_3}{\to} \ldots \overset{\varepsilon_n}{\to} m. \tag{3.14}$$

The number of edit operations in the edit history is the length *of the edit history, and we write*

$$\text{len}(p_m) := n. \tag{3.15}$$

If the true edit history p_m is not known, several edit histories are conceivable. Of special interest is then the set of the shortest edit histories

$$\text{P}(m) := \{p_m \mid \text{len}(p_m) = \min_{p_m} \text{len}(p_m)\}. \tag{3.16}$$

Similarly, for a model transformation (m_{t+1}, m_t), we define the edit history p_{m_{t+1},m_t} as the sequence of edit operations that transforms m_t to m_{t+1}, and define $P((m_{t+1}, m_t))$ similarly.

Remark 3.5.1
In the language of assembly theory, an edit history is called an assembly pathway*.*

To derive $P(m)$ for a given $m \in \mathcal{M}$ and $E \subset \mathcal{E}$, we can either use a brute force approach (i.e., trying all possible edit histories up to a certain size) or try to match large edit operations in the model m. A critical pair analysis [223] can then be used select compatible edit operations to construct an edit history. Anyway, in practice, these approaches will rarely be computationally feasible. We would rather employ heuristics to approximate $P(m)$, for example, we can start by matching large edit operations, compute the remaining difference, and then match smaller edit operations, and so on.

We can then define a conditional probability measure:

Definition 3.5.4 Conditional Edit History Probability Measure.
Given a set of edit operations E (including special markers for α, Ω for the start and end of an edit history), two model transformations $\varepsilon \in E$ and $\kappa \in E$, let $\{t_i\}_{i \in I}$ be an enumeration of all successive pairs of models in the model history, let $s_i := |P(t_i)|$ be the size of the set of the shortest edit histories for t_i given E, and let count (ε, p) *be the number of occurrences of the edit operations ε in an edit history p of a model transformation. We define a conditional probability measure*

$$\mathbb{P}(\varepsilon|\kappa) := \frac{\sum_{i \in I} \frac{1}{s_i} \sum_{p \in P(t_i)} \text{count}\,(\varepsilon\kappa, p)}{\sum_{i \in I} \frac{1}{s_i} \sum_{p \in P(t_i)} \text{count}\,(\kappa, p)}. \tag{3.17}$$

$\mathbb{P}(\varepsilon|\kappa)$ is the relative number of occurrences of the edit operation ε in the model history given that the edit operation κ has occurred—averaged across all conceivable (i.e., minimal) edit histories.

Remark 3.5.2
Note that in Definition 3.5.4, we can replace $\kappa \in E$ by any composition of edit operations in E.

Theorem 3.5.1
$\mathbb{P}(.|\kappa)$, as defined in Definition 3.5.4 defines a probability measure for every κ that can be constructed from edit operations in E.

Proof. We have to show the so-called Kolmogorov axioms, that is,

1. $\mathbb{P}(\varepsilon|\kappa) \in [0,1]$ for all $\varepsilon, \kappa \in E$,
2. $\sum_{\varepsilon \in E} \mathbb{P}(\varepsilon|\kappa) = 1$ for all $\kappa \in E$,
3. For $E_1, E_2, E_3, \cdots \in E$ with $E_i \cap E_j = \emptyset$ for $i \neq j$, we have

$$\mathbb{P}(\cup_i E_i|\kappa) = \sum_i \mathbb{P}(E_i|\kappa).$$

The first axiom is simple to verify, since κ appears more often then $\epsilon\kappa$, and nominator as well as denominator are non-negative. The second axiom follows since we can take the sum over all edit operations in E that follow κ. Since E is a generator and the end of an edit history is marked by $\Omega \in E$, the sum will be 1. The third axiom follows by definition, that is, for a set $E_i \subset E$, we can define $\mathbb{P}(E_i|\kappa) := \sum_{\varepsilon \in E_i} \mathbb{P}(\varepsilon|\kappa)$ as the sum of the individual probabilities, which are independent in the sequential edit history definition. □

We next utilize this probability measure to tackle the shortcoming of a probability measure such as the one from Equation 3.13. Given (m_{t+1}, m_t), as before, we can consider the set of the shortest edit histories $P((m_{t+1}, m_t))$. We can then define a probability measure for every model transformation (m_{t+1}, m_t):

Definition 3.5.5 *n-Edit Operation Generative Model.*
Let (m_{t+1}, m_t) be a model transformation, and let $E \subset \mathcal{E}$ be a set of edit operations, and $\mathbb{P}(\epsilon|\kappa)$ be the conditional probability measure as defined in Definition 3.5.4. For a single edit history $p = \varepsilon_k \varepsilon_{k-1} \dots \varepsilon_2 \varepsilon_1 \in P((m_{t+1}, m_t))$, we define the probability measure

$$\mathbb{P}(p) := \mathbb{P}(\varepsilon_k \varepsilon_{k-1} \dots \varepsilon_2 \varepsilon_1) \tag{3.18}$$

$$= \prod_{i=1}^{k} \mathbb{P}(\varepsilon_i|\varepsilon_{i-1} \dots \varepsilon_1) \tag{3.19}$$

Assuming a Markov chain of order n as a probabilistic model for the edit history, we can then approximate the probability measure by

$$\mathbb{P}_{Markov}(p) \approx \prod_{i=1}^{k} \mathbb{P}(\varepsilon_i | \varepsilon_{i-1} \dots \varepsilon_{i-n}). \tag{3.20}$$

For a model transformation (m_{t+1}, m_t), we define the probability measure then as the aggregation

$$\mathbb{P}^{(\text{agg})}_{Markov}((m_{t+1}, m_t)) := \text{agg}_{p \in P((m_{t+1}, m_t))} \mathbb{P}_{Markov}(p), \tag{3.21}$$

where we can either choose $\text{agg} := \text{mean}$ *or* $\text{agg} := \max$.

Still, even for small n-th order Markov chain approximation, this model has several computational difficulties:

Many Histories Challenge Especially for large model transformations t_i, if we do not know the true edit history, the set $P(t_i)$ will be very large, and the computation of $\mathbb{P}^{(\text{agg})}_{Markov}(t_i)$ will be computationally expensive.

Curse of Dimensionality The number of possible edit histories grows exponentially with the number of edit operations in the set E and the number of edit operations in the edit history.

Computational Complexity Given a set of edit operations E, deriving the edit histories $P(t_i)$ is computationally expensive.

Unavailability of E In many cases, a sufficient model E is not known or only *atomic* edit operations can be derived from a meta-model.

Approximation Error The Markov chain approximation is only an approximation and might not be accurate.

Data Sparsity or Out-of-Vocabulary Issue The number of observed edit histories is typically very small compared to the number of possible edit histories. For larger values of n (i.e., the order of the Markov chain in the approximation), we will typically encounter combinations of edit operations that have not been observed in the data set.

Even if some mechanism to derive edit operations is available (e.g., edit operation recording [131]), we will be challenged by the data sparsity issue. Indeed, as discussed in Section 3.4.2, these challenges are very similar to the challenges faced in statistical language modeling. In language models n-gram models that have been used for a long time and faced similar challenges as the data sparsity issue [148]. In fact,

assuming natural language labels in the models, the problem described here is a super class of the problem of language modeling.

Theorem 3.5.2
The problem of deriving a generative model for language is a subclass of the problem of modeling software models.

Proof. We can define a meta-model with one class word and a linked list structure, that is, every word has a predecessor. E can be defined by comprising the set of all edit operations to add a word in the language to a sequence of sentences. Every text would then be a model according to this meta-model, and the (generative) process of writing would be a sequence of edit operations. □

This analogy shows that (except for domains with a very limited label alphabet), utilizing the generative model from Definition 3.5.5 will face similar problems as n-gram language models. The solution in the area of language modeling has been to use parametric models, in particular, neural networks. It is therefore conceivable that also for software models, a parametric model is more likely to be a feasible approach to capture the regularities in historic modeling data.

Hypothesis

Hypothesis 2. Parametric approaches are likely a more feasible approach (from a computational perspective) to model the generative process of software models, compared to pattern-based approaches, and, in particular, for use cases such as software model generation or software model completion.

The most typical use case for a generative model for software models is model (auto-)completion (similar to code completion for generative models for source code).

Example 3.5.2 Generative Models for Software Model Completion.
A generative model $\mathbb{P}(\epsilon|\kappa)$ as in Definition 3.5.4 can be used as a solution to the model completion problem Section 3.5.3. Given a history of model transformations, we can sample from the generative model to obtain a model transformation (m_{t+1}, m_t). For example, given a model transformation $m_{t-1} \xrightarrow{\varepsilon} m_t$, we can select γ with the highest probability according to $\mathbb{P}$, that is,

$$\gamma := \arg\max_{\gamma} \mathbb{P}(\gamma|\varepsilon).$$

Anyway, for some use cases, a generative model is not sufficient—we want to have more explicit patterns at hand. For example, we might want to have a high-level description of a large change from one model revision to the next. Also for theoretical purposes, the relationship between explicit patterns and implicit parametric genera-

tive models is of interest. We therefore have to conduct experiments that investigate the boundary between explicit edit operations and parametric models, such as large language models and graph neural networks in Chapter 7.

Edit Operation Mining from Model Histories

In the construction of model completion above, we assumed that a set of edit operations E is known for the construction of a generative model. In this section, we discuss how such a set of edit operations can be derived from the model history in a data-driven manner. Of course, in order to be of any use for theoretical and practical purposes, edit operations in E should "make sense" or "be meaningful". We will show that our definition of "meaningfulness" is related to a measure for "the amount of selection" in Assembly Theory [295] published shortly after our publication of edit operation mining [324].

In Section 3.5.2 we formally defined the use case of edit operation mining (see Definition 3.5.1), that is, given $M \subset \mathcal{M}$, where $\mathcal{M}$ adheres to a meta-model $\mathcal{TM}$, we want to derive a set of edit operations $E \subset \mathcal{E}$.

We have seen that the challenge of this task is to find a trade-off between two extremes:

- "Small" and frequent edit operations, and
- complex but infrequent, often too specific edit operations.

Just frequent edit operations are usually not "meaningful", for example, because for every edit operation all of its parts will also be frequent, but might not be complete. Too complex edit operations, on the other hand, might be rather individual model parts that will not be reused.

As in Example 3.2.11, a solution to this trade-off is to use the Minimum Description Length Principle (cf. Section 3.2.2). Applying the Minimum Description Length Principle, we would choose E such that our observation of models $M \subset \mathcal{M}$, is compressed the most, that is, we would choose E such that

$$L(M) = \min_{E \in 2^{\mathcal{E}}} L(M|E) + L(E). \tag{3.22}$$

Of course, what we only know when looking at the collections of models M is that E, as defined by equation 3.22, explains a bias that we can observe in the data *after the fact*—there is no guarantee that the tool also provides these edit operations. Furthermore, adopted patterns and adopted edit operations provided by a tool will lead to a large "copy number" and therefore a bias in the data. Anyway, an edit operation $\varepsilon \in E$, as defined by equation 3.22, might not be "meaningful" to a tool user and not perfectly coincide with adopted patterns and edit operations.[26]

26 To formulate this more formally, we denote by E^* a set of operations that would be considered meaningful and helpful (and therefore would be adopted) by a tool user, and by E the set of edit operations that are most compressing according to the MDL principle. Because of the adoption, $L(M|E^*) + L(E^*) \leq L(M)$. Anyway, it could still be that $L(M|E) + L(E) < L(M|E^*) + L(E^*)$.

We assume that this is not the case, that is, we formulate the following hypothesis:

Hypothesis

Hypothesis 3 (Inference of Meaningful Edit Operations)**.** The most compressing set of edit operations E, according to the Minimum Description Length Principle (Principle 3.2.1), is a set of edit operations that is meaningful and useful for a tool user.

This hypothesis

- is a concrete application of Occam's razor to the modeling process,
- is related to selection as described by Assembly Theory (as we will see below),
- can not be proven formally, since it relates social concepts without any formal theory such as "meaningfulness" to a mathematical concept (i.e., the Minimum Description Length Principle), and
- has therefore to be proven empirically.

Remark 3.5.3
$L(M|E)$ is mainly driven by three factors: the number of edit operations in E, the complexity of the "interface" of the edit operations in E, and how many edit operations are applied to create M from E. Roughly speaking, this means that $L(M|E)$ is related to the effort the user has to put in to create M from E, that is, the effort for looking up every applied edit operation, and inserting the correct parameters. This gives another viewpoint of the "meaningfulness" of E in MDL. E, minimizing Equation 3.22, is a set of edit operations that keeps the effort low to construct M from E, while also keeping the complexity of the descriptions of E low.

Example 3.5.3
We extend our Example 3.3.1 from Section 3.3.1. In Figure 3.9, we see a model history M and a pattern subgraph $g_1 \in E$. The compressed model history $M|E$ can be constructed by replacing the occurrences of g_1 in M by a reference to g_1. The subgraph g_1 is a pattern that appears three times in the model history M. We can therefore compress the model history M by replacing the occurrences of g_1 by a reference to g_1. Therefore, the description of the pattern E plus the description of the model history by means of the patterns (that represent our candidate edit operations) $M|E$, is typically smaller than the original description M.

In practice, finding an E that minimizes Equation 3.22 is computationally infeasible. Even matching edit operations in the model history is computationally

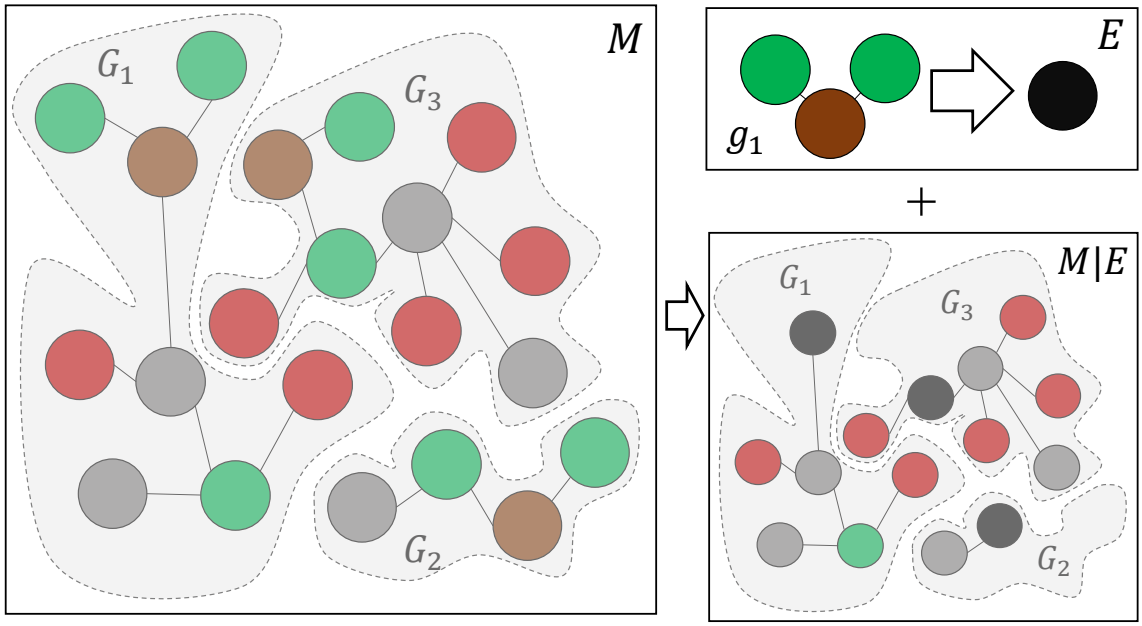

Figure 3.9: Example of a model history M, the definition of a pattern subgraph $g_1 \in E$ (i.e., edit operation, in the case of difference graphs), and the compressed model history $M|E$. Since we can now reuse the pattern subgraph g_1 three times, but only need to store it one time, the description of the pattern E plus the description of the model history by means of the candidate edit operations $M|E$, is smaller than the original description M. Of course, given E, there is still some freedom in how to describe the model history $M|E$, for example, patterns might overlap, the order of the execution is not fixed. Furthermore, the application of the edit operation requires additional information such as parameters (e.g., attribute values)—an interface for the application of edit operations is necessary.

expensive, as we elaborated on in Section 3.3. In practice, one therefore has to rely on heuristics to approximate E.

There are two main ideas that we will investigate in this thesis:

Graph Mining The first approach employs graph mining techniques to list individual pattern candidates and rank them.

Generative Models The second approach trains a generative parametric model and tries to derive patterns from the generative model.

In the graph mining approach, we first discover frequent[27] subgraph patterns g in change graphs. One can then rank the candidates according to their capability of compressing the data. Concretely, we will work with a heuristic compression measure

27 There are also compression based miners such as Subdue [72]. In pilot experiments they did not perform well for our data though.

$$\text{compression}(g) := \frac{\text{size}(g)(\text{supp}(g) - 1)}{\text{size}(M)}, \tag{3.23}$$

where the size of a graph can be approximated by the number of nodes plus the number of edges, and supp is the support of the pattern in the model history M. The higher compression(g) is for a frequent pattern, the higher it will be ranked in the list of patterns.

In section Section 3.2.2, in Equation 3.6, we introduced the quantity A from Assembly Theory. The authors relate this quantity to "the amount of selection". We will now see that this quantity A is tightly related to the compression measure from Equation 3.23.

Theorem 3.5.3
Suppose for the distribution of the size *of subgraphs g with assembly index a_g, we have*

$$\mathbb{E}_{a_g=t}(\text{size}(g)) = c(t-1)\mathbb{E}_{a_{g'}=t-1}(\text{size}(g')),$$

where $1 < c(t-1) \leq 2$ for every $t \in \mathbb{N}$.
Then there are constants α_0, α_1, c_0, and c_1 such that, in expectation, the compression of a subgraph g with assembly index a_g is bounded by

$$c_0\mathbb{E}(e^{\alpha_0 a_g}(n_g - 1)) \leq \mathbb{E}(\text{compression}(g)) \leq c_1\mathbb{E}(e^{\alpha_1 a_g}(n_g - 1)),$$

that is, for

$$A_g^\alpha := e^{\alpha a_g}(n_g - 1),$$

we have

$$c_0\,\mathbb{E}(A_g^{\alpha_0}) \leq \mathbb{E}(\text{compression}(g)) \leq c_1\,\mathbb{E}(A_g^{\alpha_1}).$$

Proof. The copy number n_g is the number of occurrences of a subgraph g in the model history M, that is, the support of the subgraph. Furthermore, since $size(M)$ is a constant, by comparing compression(g) with A_g, what remains to show is that there are constants c_0, c_1, α_0, and α_1 such that

$$\begin{aligned} & c_0\,\mathbb{E}(e^{\alpha_0 a_g}) \leq \mathbb{E}(\text{size}(g)) \leq c_1\mathbb{E}(e^{\alpha_1 a_g}) \\ \Leftrightarrow \quad & c_0\,e^{\alpha_0 a_g} \leq \mathbb{E}(\text{size}(g)) \leq c_1\,e^{\alpha_1 a_g}. \end{aligned}$$

Because of the assumption that, in expectation, the size of subgraphs with assembly index t increases by a factor of $c(t-1)$, compared to the size of subgraphs with assembly index $t-1$, by induction, we have

$$\begin{aligned}\mathbb{E}(\text{size}(g)) &= c(t-1)\mathbb{E}(\text{size}(g')) \\ &= c(t-1)c(t-2)\mathbb{E}(\text{size}(g'')) \\ &= \cdots = \Pi_{i=1}^{t-1} c(i)\mathbb{E}(\text{size}(g_0)),\end{aligned}$$

where g_0 is an initial building block. The term $\mathbb{E}(\text{size}(g_0))$ we can move into the constants c_0, and c_1. For the rest, we define $c_{min} := \min c(i)$ and $c_{max} := \max c(i)$. We then have

$$\begin{aligned}&\mathbb{E}(\text{size}(g_0))\, c_{min}^{(t-1)} \leq \mathbb{E}(\text{size}(g)) \leq \mathbb{E}(\text{size}(g_0))\, c_{max}^{(t-1)} \\ \Leftrightarrow \quad &\frac{\mathbb{E}(\text{size}(g_0))}{c_{min}}\, c_{min}^{a_g} \leq \mathbb{E}(\text{size}(g)) \leq \frac{\mathbb{E}(\text{size}(g_0))}{c_{max}}\, c_{max}^{a_g}\end{aligned}$$

Using that

$$a^x = b^{x\frac{\ln(a)}{\ln(b)}},$$

and $\ln e = 1$, we have

$$\frac{\mathbb{E}(\text{size}(g_0))}{c_{min}}\, e^{\ln c_{min}\, a_g} \leq \mathbb{E}(\text{size}(g)) \leq \frac{\mathbb{E}(\text{size}(g_0))}{c_{max}}\, e^{\ln c_{max}\, a_g}$$

With $\alpha_0 := \ln c_{min}$, $\alpha_1 := \ln c_{max}$, $c_0 := \frac{\mathbb{E}(\text{size}(g_0))}{c_{min}}$, and $c_1 := \frac{\mathbb{E}(\text{size}(g_0))}{c_{max}}$ we have

$$c_0\, e^{\alpha_0\, a_g} \leq \mathbb{E}(\text{size}(g)) \leq c_1\, e^{\alpha_1\, a_g},$$

that is, the relation we wanted to show. □

Corollary 3.5.1
Suppose $c(t) := c$ is constant then we have

$$\mathbb{E}(\text{compression}(g)) = c'\, \mathbb{E}(e^{\ln c\, a_g}(n_g - 1)),$$

where c' is the normalization constant that depends on the initial building block g_0, c, $\text{size}(M)$, and N_T.

As in corollary 3.5.1, we can see that the compression of a subgraph g is related to the assembly index a_g and the copy number n_g.

Proof. The proof follows directly from the proof of Theorem 3.5.3, with

$$c_{min} = c_{max} = c.$$

□

Recall the formula for the assembly index A from Equation 3.6:

$$A = \sum_g e^{a_g} \left(\frac{n_g - 1}{N_\mathrm{T}} \right).$$

The term A_g^α in Theorem 3.5.3 is therefore equal to a (scaled) contribution of g to the assembly A.

Remark 3.5.4
Regarding the scaling factor $\ln c$ *in the exponential above, we believe it should rather be added to the definition of* A*, that is, it is a parameter of* A *itself and in this sense,* A *and* $\mathrm{compression}(g)$ *measure the same quantity.*

Having a $c > 1$, that is, c strictly greater than 1, means it is more likely that objects are assembled from more complex objects than from less complex objects. This is also very intuitive for the assembly of software: Even though conceivable, it is less likely that we combine something on a very "recent" level ($\sim$ assembly index) with something on an "older" level. For example, we combine python libraries in python code, but we rarely combine python code with machine code.

In the extreme case of always combining two objects from the most recent level to form a new object, we would have $c = 2$.

Summary
In this section, we have defined two tasks, edit operation mining in software model histories and software model completion, which are directly related to the evolution of single but also an ecosystem of software models. Edit operations have been hypothesized to be patterns in the model history that are meaningful and useful for a tool user. We have also seen, how a generative model for software models in a certain ecosystem is related to these patterns: These operations can be used to sequence the models in the ecosystem, and an n-Edit Operation Generative Model (similar to n-gram models for natural language), can be defined to approximate the generative process of software models. Because of data sparsity issues,[28] we have hypothesized that a parametric model is more likely to be feasible than a generative model based on counting the occurrences of edit operations in the data set.

We have made the egregious hypothesis that it is the most compressing set of edit operations that is meaningful and useful for a (modeling) tool user. This hypothesis can be motivated through two lenses—an *a priori* lens and an *a posteriori* lens:

A Priori Lens Possible combinations (assemblies) underlay selection. Only the selected combinations will achieve high abundance (i.e., a large copy number). Selection leads to a pressure on the assembly index that leads to objects of large

28 The computational complexity for subgraph mining is also a limitation of "exact" solutions to the problem.

assembly index with a large copy number. The amount of "selection" necessary to favor high assembly index objects depends exponentially on the assembly index and linearly on the copy number [295].

A Posteriori Lens Observing a set of assembled objects, we can decompose objects in constituent sub-objects, subsystems, edit operations, etc. The decomposition of the set of objects giving rise to the most compressed description of the set of objects defines meaningful sub-objects, subsystems, edit operations, etc.

This basically means, one either takes a standpoint from the current point in time and considers selection to be the explanation for the objects in an ecosystem, or one looks into the past and defines objects in a way to minimize the description of the observation. It is Theorem 3.5.3 that shows that both viewpoints give rise to the same definitions of objects.

Part II

Case Study & Thesis Context & Datasets

4

A Railway Industry Case Study

Simplicity does not precede complexity, but follows it.

— Alan Perlis

This chapter shares material with the Foundation of Software Engineering (FSE 2022) conference paper [325].

In this chapter, we present a case study of a large-scale industrial model-driven managed cloning software product line in the railway domain at Siemens Mobility. On the one hand, this case study serves as a motivation for the research presented in this thesis by providing a concrete example of the complexity of a real-world industrial model-driven product line. On the other hand, we use data, that is, the system models from this case study to empirically evaluate approaches for feature mining in this chapter, edit operation mining in Chapter 6, and software model (auto-)completion in Chapter 8. The context provided in this chapter is essential to understand the complexity of engineering efforts behind the dataset.

Many industrial software product lines use a clone-and-own or managed cloning approach for reuse among software products. As a result, the different products in the product line may *drift apart*, which implies increased efforts for tasks such as change propagation, domain analysis, and quality assurance. While many solutions have been proposed in the literature, these are often difficult to apply in a real-world setting. We study this drift of products in a concrete large-scale industrial model-driven managed cloning software product line in the railway domain at Siemens Mobility. We demonstrate that extracting features from the set of products is not easily feasible in the project presented in this chapter, as the products are too complex and there is a lot of noise in the data. For this purpose, we conducted interviews and a survey, and we investigated the models in the model history of this project. In the interviews and survey conducted for this product line, we found that increased

efforts are mainly caused by large model differences and increased communication efforts. Furthermore, we investigate a feature mining approach to the modeling data in this project and find that the complexity and noise in the data make approaches like these unlikely to be successful, in practice. We show that feature mining can be formulated as an application of the Minimum Description Length principle from Section 3.2.2.

We argue that, in the short-term, treating the symptoms (i.e., handling large model differences) can help to keep efforts for software product-line engineering acceptable—instead of employing sophisticated variability management. To treat the symptoms, we employ a solution based on semantic-lifting to simplify model differences. Using the interviews and the survey, we evaluate the feasibility of variability management approaches and the semantic-lifting approach in the context of this project.

The setting and challenges presented in this chapter have been the main motivation for the research on Intelligent Modeling Assistants in this thesis. Indeed, we find that tool support, especially for support in evolution-related tasks such as change propagation, is heavily demanded, since manual change propagation and necessary communication efforts are costly.

4.1 Drift in Software Product Lines

Software product-line engineering [259] aims at developing a family of software products by reusing software artifacts as well as processes across the family. There are several software product-line engineering methodologies [16, 274], ranging from simple cloning of artifacts across products (clone-and-own) to platforms with built-in variability support, which allow to generate products based on a configuration (i.e., a feature selection). Depending on various factors (e.g., the number of products), it makes sense to favor one approach over another [86, 190, 259, 274]. For example, the overhead for establishing and using a platform will only pay off when the platform is used for a larger number of products [259]. It is important to note that the reuse of artifacts in software product lines is not limited to source code, but can also include other artifacts, such as documentation, requirements, and models [17, 18, 30, 226, 256, 335].

Many industrial software product lines have been grown organically, using some form of managed cloning (clone-and-own with some form of clone management in place) [86, 274]. As a consequence, changes have to be propagated from one product to another. This is often done manually and therefore time-consuming and error-prone [190]. A key challenge of the evolution of clone-and-own software product lines is that products tend to drift apart [167, 363]. This leads to a risk of losing the benefits of reuse if efforts related to cloning and merging exceed the savings of reuse.

In this chapter, we report on a model-driven software product line (i.e., models are the primary artifacts) in the railway domain at Siemens Mobility, which employs a form of managed cloning. In particular, we study the divergence of models in this setting. We observed that quality assurance, change propagation, and domain analysis are the main drivers for increased effort (or costs), due to the cloning approach—an observation consistent with the literature [86, 190, 275]. Nevertheless, these activities are indispensable. Especially for safety-relevant parts of the system, a thorough quality assurance of the changes made to the system is critical to decrease the risk of failing software qualification, which in turn leads to an increase in time-to-market.

Common to all of these activities is the analysis of *differences* between products. In large software product lines, the models and also the differences between models easily become huge. We found that, indeed, large model differences are a major symptom of the drift at Siemens Mobility. It is increasingly difficult to identify commonalities that can be lifted to a common platform or propagated across products. Many small but uncritical changes, such as model element renamings, make model differences unnecessarily verbose and confusing. For example, in our case study, renaming a package leads to the change of the fully qualified name of subordinated elements and therefore to plenty of fine-grained changes in the model difference. A solution to the divergence problem might be to migrate to a platform-based approach [86, 190, 274]. This is not only a time-consuming task, but there is also a risk of losing flexibility to create new products. Moreover, the independence of products in a clone-and-own software product line is sometimes desired to avoid unintended side effects [86, 190]. In general, often assumptions made by platform approaches are not compatible with real-world constraints such as organizational structures within a company. An alternative to platform-based approaches is to utilize feature traces in a bottom-up manner for change propagation and the composition of new products [98, 167]. Unfortunately, state-of-the-art tools are not mature yet (e.g., scalability to large product lines has not been shown yet) and are not available for model artifacts.

We will apply a feature identification approach based on the Minimum Description Length principle to a set of products to show the difficulties and limitations that hinder semi-automated migration to a platform or bottom-up approach.

While it is desirable in the long term to systematically manage variability, it is imperative in the short term to "treat the symptoms" of a clone-and-own approach. That is, we need to reduce the effort required for critical activities such as quality assurance, change propagation, and domain analysis. For this purpose, we have combined semantic lifting [163] with an approach for edit operation mining [324], to effectively simplify large model differences at Siemens Mobility. The idea of this semantic lifting approach is to *compress* the differences, that is, to combine many fine-grained differences into higher-level, reusable change patterns. The theory behind this approach has been laid out in Section 3.5.4. We will discuss this approach in more detail in Chapter 6.

Based on semi-structured interviews and a questionnaire survey, we explore the merits and barriers of platform-based approaches, bottom-up approaches, and the semantic lifting approach in the context of drift between products in an industrial setting.

The results of the analysis of the particular case study highlights that several challenges in model-driven and model-based software product line, in particular, (1) organizational challenges, and (2) tool challenges to support the "evolution" of a single product as well as the evolution of the entire product line. For this thesis, we focus on the tool challenges aspect, although one has to keep in mind that the tooling needs to take the organizational setup and organizational constraints into account.

4.2 Software Product-Line Engineering: Evolution in Space

A train from New York Central Station to Broadway would very likely be different from a train from New York to Boston Massachusetts. On a journey from New York to Boston, a customer would expect sanitary facilities, while on a journey to Broadway, the customer could use the sanitary facilities at the subway stations. Although there are many differences, there are also similarities between the two trains. For example, the controls for the interior lighting, or the air conditioning might be very similar.

To cope with different application scenarios and customer requirements, many modern (software) systems are developed as a software product line. Roughly speaking, this means design, engineering, and production of the different products addressing diverse customer requirements are not performed individually. Instead, there is a development and production of a product line that includes the commonalities of different products and variability among them. In this section, we will give an overview of software product-line engineering.

4.2.1 Software Product-Line Engineering: Basics

In this section, we will provide the basic terminology of software product-line engineering.

Definition 4.2.1 Software Product Line – Software Engineering Institute [70].
A software product line *is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.*

Remark 4.2.1
Some authors use the term software product family *as a synonym for* software product line.

Software product-line engineering is the engineering discipline that deals with the development of software product lines.

Definition 4.2.2 Software Product-Line Engineering – adapted from Krueger and Clements [188].
[Software] Product-Line Engineering *refers to the disciplined engineering of a portfolio of related products using a common set of shared assets and a common means of production [and distribution].*

The idea of software product-line engineering probably explicitly appeared for the first time in James Neighbors 1980 dissertation "Software Construction Using Components" [239], where he introduced the concept of developing a family of related software products from a shared set of assets. In the 1980s and 1990s, the idea of mass customization emerged, paralleling Software Product-Line Engineering's goal of efficiently delivering tailored software solutions through systematic reuse. The Software Engineering Institute (SEI) formalized Software Product-Line Engineering during this period, defining it as a set of software-intensive systems that share a common, managed set of features to meet specific market needs. Key publications such as "Software Product Lines: Practices and Patterns" [70] by Clements and Northrop, "Software Product Line Engineering: Foundations, Principles, and Techniques" [259] by Pohl et al., and "Feature-Oriented Software Product Lines - Concepts and Implementation" [16] by Apel et al. have significantly contributed to the field, outlining best practices and methodologies.

Many concrete approaches to software product-line engineering distinguish between two main processes: *domain engineering* and *product (or application) engineering* [16, 259]:

Definition 4.2.3 Domain Engineering and Product Engineering.
Domain engineering *is the process of identifying, designing, and implementing the commonalities and variabilities of a product line. The purpose of domain engineering is to develop a set of reusable assets that can be used to create a family of products.*
Product engineering *or* application engineering*—in the area of software development—is the process of creating a specific product from reusable assets (and possibly adding product specific features). Product engineering comprises the activities (or phases) of* requirement analysis, product design, product implementation, *and typically also some validation and verification activities.*

4.2.2 State of the Art

Whereas model evolution research [171, 328] is focusing on the evolution in the *temporal dimension*, the software product-line engineering community is also interested in differences in the *product dimension*. In the literature, several alternatives and approaches complementary to clone-and-own are proposed. In what follows, we will recapture the clone-and-own approach, describe the problem of the so called *unintentional divergence* in clone-and-own, and describe possible solutions proposed in the literature. Furthermore, we provide an overview of case studies about clone-and-own in an industrial setting.

Clone-and-Own: Clone-and-own [86, 167, 190, 274, 275] describes the *ad-hoc* process of cloning existing products and developing the cloned products *independently* of each other. There has been a debate in the software product-line engineering community about whether a clone-and-own approach should be discouraged or not [86, 156, 167]. Nevertheless, as a matter of fact [86, 167, 274], clone-and-own approaches are used in the industrial practice. In many cases utilizing clone-and-own approaches, there exists a single product which has to be modified slightly to satisfy the needs of other customers. Cloning is then an available mechanism, which can be rapidly used to satisfy this customers needs. Often it is not known in advance whether an existing product will evolve into a family of products. Therefore, it is hard to assess whether the initial effort to setup a platform for the product family will pay off in the long run [259]. An observation made by Berger et al. [34] is "that none of our subjects exercises pure clone&own, but that variants already use variation points". This observation is also true for our railway case study, where trainset platforms also have some internal variability (see Section 4.3).

A common issue related to the clone-and-own approach is the unintentional divergence between the products [167, 286]. *Unintentional divergence* describes the increasing difference between products in the product family. There are multiple possible reasons for this divergence, for example, missing reuse opportunities (e.g., commonalities are not identified during domain analysis), *noise* due to differences in the implementation of features in different products, or even the increased complexity of the individual products as a consequence of the evolution of the products. In the context of a model-driven product line, we will refer to *model drift* as a specific instance of unintentional divergence in model artifacts.

Platform Approaches: One solution to the divergence problem is to migrate the cloned product line to a *150% platform* [86, 190, 274]. There are only a few tools that support 150% platform approaches for model-driven software product lines. SUPERMOD [290] relies on so-called *filtered editing* and *feature ambitions*, which a user can select when committing changes. The MAGICDRAW PRODUCT LINE ENGINEERING PLUGIN is more mature, but the user has to explicitly model the variability for every

(variable) model element. In contrast to 150% platform approaches, where a product is generated by selecting a subset of the functionality implemented in the 150% platform, compositional approaches [17, 94, 147, 257, 282] have been proposed in the literature. For example, Apel et al. [17] propose an approach to compose a specific product by superimposing *model fragments*; typically one model fragment per feature. Jayaraman et al. [147] describe a UML composition language based on graph transformations for reusing features. Another compositional approach is *delta modeling* [282], where a product is composed out of a *core model* (a valid product) and multiple delta-models that encapsulate modifications to the core model. Delta-models are not necessarily in one-to-one correspondence with features. Elrad et al. [94] propose an approach for aspect-oriented modeling to separate crosscutting-concerns and core functionality in UML models.

A migration from a clone-and-own approach to a platform approach is not only time-consuming, but there is also a risk of losing flexibility to create new products, and the independence of the products in a clone-and-own software product line is sometimes desired to avoid unintended side effects [86, 190].

Bottom-up Product Line Engineering: Some authors observe that, with an increasing number of products, maintenance efforts for a clone-and-own approach will rapidly grow [86, 167, 254, 358], but they also acknowledge that, in some settings (e.g., a few products), a cloning approach is viable [156, 167, 254, 259]. Furthermore, many industrial product lines start with a single product, and it is not clear at the beginning, how many products there will be in the future or how long the product line will be used at all [86, 163].

For this reason, approaches supporting variability management in clone-and-own product lines have been proposed [98, 167]. The basic idea is to utilize feature traces in a bottom-up manner for change propagation and the composition of new products. Some authors propose *variation control systems* [98, 157, 207, 290, 308], which aim at unifying version control and variability management.[1]

Industrial Clone-and-Own Case Studies: Clone-and-own has been studied in multiple-case studies (i.e., having more than one concrete subject project) and single-case studies.

For example, Rubin et al. [274] evaluate a framework for managing collections of related products based on three industrial case studies. Dubinsky et al. [86] study cloning practices in six industrial clone-and-own software product lines. More recently, Krüger et al. [190] compare cost and cost factors of clone-and-own and platform-oriented reuse in industrial settings using a literature review and interviews within one company. Berger et al. [34] study the state of adoption of a systematic variability management in twelve industry case studies across several domains. While

1 Some variation control systems internally maintain 150% model, and therefore belong to the category of platform approaches.

this research compares platform-oriented research and clone-and-own or investigates a transition from clone-and-own to a more systematic variability management, in this chapter, we study the concrete challenges related to the evolution and drift between products in one concrete industrial clone-and-own software product line.

Regarding single-case studies in an industrial context, for example, Jepsen et al. [149] or Weston et al. [344] focusing on the extractive adoption of software product lines. Other single-case studies compare platform-based and clone-and-own approaches, for example, Echeverría et al. [88] compare effectiveness, efficiency, and satisfaction of clone-and-own and a platform-based approach in a controlled experiment. A comprehensive list of case studies using extractive adoption is given by the *ESPLA catalog* [213]. To the best of our knowledge, there is no work studying drift between products in an industrial setting.

Variation Control Systems and Filtered Editing: A very active research topic is the unification of version and variability management or "software evolution in time and space" [33]. A step towards this goal are so-called variation control systems [98, 157, 207, 290, 308], for example, ECCO [98], which supports the feature extraction and product composition of clone-and-own based software product lines, or SUPERMOD [290], which uses so called filtered editing to support collaborative editing of model-driven product lines. In these tools, a unified platform (150% platform) of all products is maintained and given a feature selection (or configuration), a filtered view on this unified platform is presented to the tool user. Variability related changes could be hidden or highlighted (depending on the task) in the model using approaches as for example filtered editing [290]. A selection of relevant features for a specific analysis will decrease the model complexity and cognitive overhead for several product and domain engineering-related tasks such as change propagation (see Figure 4.1). Only SUPERMOD currently provides explicit support for models.

Even though these tools would ease the challenges described above, they are often research tools and their scalability to industrial application is unclear, or they have other functional shortcomings, for example, missing support for SysML. Also, a migration from MAGICDRAW to another tool would lead to large migration efforts and high costs for training the engineers using the tools as discussed in a study by Berger et al. [34]. Other platform approaches are also not feasible, because engineers want to think in terms of products not in terms of a platform and features. Furthermore, as also discussed in this chapter, there is a lack of (explicit) software product line engineering concepts in many domain specific languages. This is also true for SysML, even though SysML provides means to incorporate variability concepts into the language.

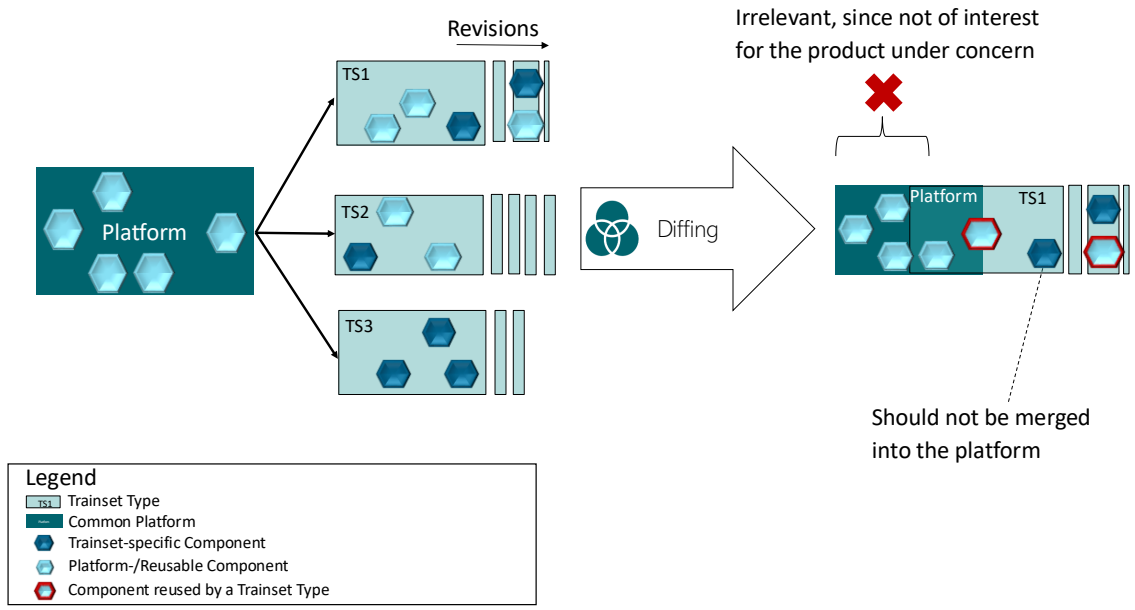

Figure 4.1: Schematic example of a difference between a platform and a product (here trainset type). Comparing platform and product can be relevant for several product line engineering activities, in particular, change propagation and domain analysis. Light blue hexagons denote reusable features, purple hexagons denote product-specific features. If features are not explicitly known, a difference between a platform and a product can be very large. For example, suppose one wants to propagate changes from a specific product (or trainset type) back to a common platform. In the example, only the changes to reusable features would be of interest. Product-specific parts or features, or features not used by the product at all, are not of interest in the corresponding merge scenario. In the model-driven world, often 3-way merge is not available and in the difference between two models, a huge amount of unrelated changes will be shown to the user.

4.3 Railway Case Study Setting

In this section, we describe the setting of our case study that we conducted with Siemens Mobility to understand the challenges in real-world large-scale software product line engineering.

Siemens Mobility offers a wide range of trains from small commuter rails to high-speed trains. Their trains include a lot of software components that provide various functionality, from heating, ventilation, and air conditioning (HVAC) systems, to highly safety-relevant systems such as the drive control system. The software components consist mainly of SCL code[2]. Over the years, Siemens Mobility's code bases

2 Sometimes referred to as structured text (ST); a block-structured IEC 61131-3 language, mainly designed for programmable logic controllers.

have become large and complex. For this reason, in 2014, the engineering team that is responsible for the train software decided to follow a model-driven engineering[3] methodology. This eases the documentation of the train software and maintains traceability between requirements and software components. The team of around 200 engineers uses MAGICDRAW [141] as their modeling environment, because it is an industry-proven tool, supports modeling in the modeling language used in this project (i.e., SysML[4]), and is capable of handling large models. As a further benefit of MDE, large parts of the software components' source code are generated out of the SysML models. For code generation, MAGICDRAW models are transformed into Eclipse Modeling Framework (EMF) models [310], which are then transformed into SCL source code components. The reason to use EMF in an intermediate step is the availability of tooling and libraries from the EMF ecosystem. In this project, code generation is a one-way process, that is, no round-trip engineering is performed.

At the time of the study, the model base consisted of approximately 300GB of artifacts (including image data for diagrams). The overall system model for the entire train software is divided into 59 weakly coupled submodels (called part systems), which are evolved and versioned independently of each other. For instance, there is one submodel for HVAC and another submodel for drive and break control.

The train domain is a highly regulated domain, so the software is subject to qualification and certification requirements according to IEC 61508 [140], EN 50126 [55], EN 50657 [57], and EN 50129 [56]. Maintaining requirements traceability and exhaustive documentation is therefore required, and following a model-driven approach helps to automate many of the documentation related tasks. In Table 4.1, we give an overview of the project setting.

Since many of the software components shall be reused across different trainsets such as commuter rail, light rail, and high-speed trains, the software is developed as a software product line. The artifacts are versioned in a repository with one branch per trainset type, called *trainset type platform*. There is another platform, called the *common platform*, which holds the type-independent parts of the models and the ones that can be configured/adjusted in the train-specific branches. The common platform

3 Note that mainly the structure of the code is generated from the models, while the behavioral aspects are not generated (but should conform to the models). Furthermore, there is no roundtrip engineering, that is, adjustments at a source code level need to be synchronized manually with the models. Therefore, the approach has to be rather classified somewhere in-between Model-based System Engineering and Model-driven Engineering. For simplicity, in this thesis, we will use the term Model-driven Engineering, to describe the development methodology.

4 SysML (Systems Modeling Language) is a general-purpose modeling language for systems engineering that supports the specification, analysis, design, verification, and validation of complex systems, including hardware, software, information, personnel, procedures, and facilities. Even though derived from UML, which is primarily focused on software development, SysML can model a wider range of systems, removes some of UML's software-centric restrictions, and includes additional diagram types (Requirements and Parametric Diagrams). This makes SysML more flexible and expressive for systems engineering applications. Note that the newer SysML v2 standard, which is not yet used here, is based on KerML and not directly on UML.

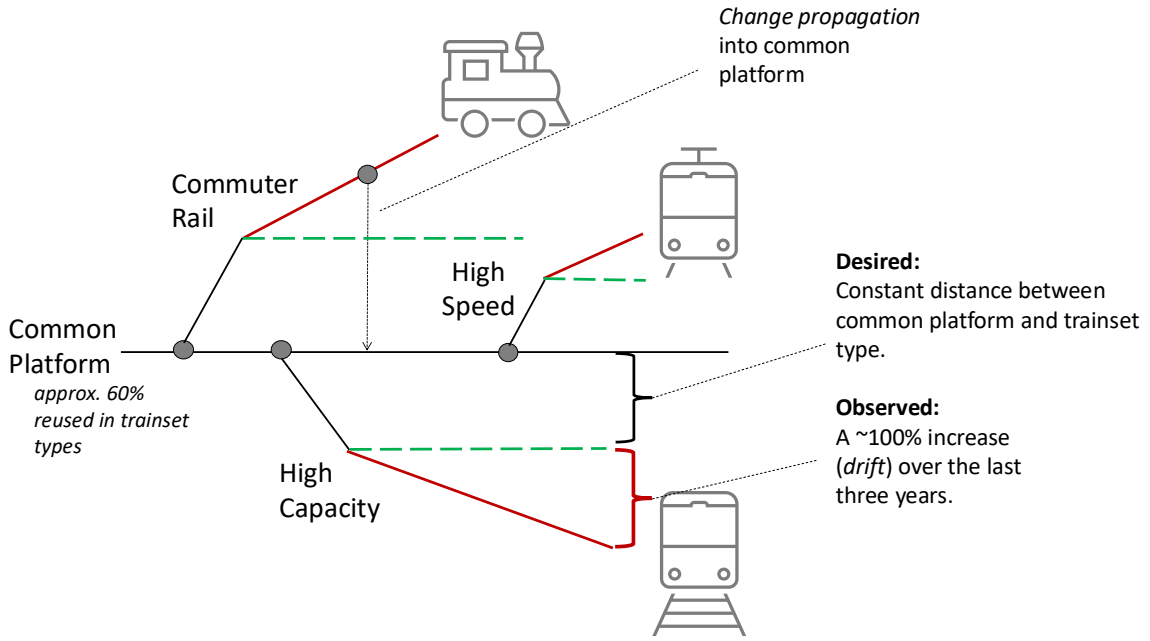

Figure 4.2: Schematic representation of model drift between trainset platforms and common platform in the railway product line.

is evolved by following an extractive software product line adoption path [16], that is, newly developed features and reusable parts are (manually) identified and regularly merged into the common platform (see Figure 4.2). Common functionality is evolved in the common platform and propagated to trainset type platforms, if desired. Furthermore, fixes or changes to the models for the trainset types have to be propagated to the common platform and trainset types that use the affected parts.

In general, *change and feature propagation* needs to be performed carefully, since parts of the software system might have been qualified already and must not be modified thereafter. Several times per month, for *quality assurance* purposes, the common platform and trainset type platform branches are compared in the modeling tool MAGICDRAW and manually checked. Incompatible or incomplete changes have to be identified during these checks. Some changes might even cause serious problems if merged into the common platform and therefore need to be identified and fixed. Furthermore, based on the model differences, a *domain analysis* is performed by a domain expert to identify commonalities and reuse opportunities.

During these change and feature propagation activities, engineers perceived a drift (see Figure 4.2), which we want to analyze further in this chapter.

Table 4.1: Project context for our railway case study. The model size is given in number of model elements (attributes not included). Model differences are between a trainset platform and the common platform and include changed attributes.

Number of developers	Train types	Sub-models	Avg. model size	Avg. number of differences	Artifact size	Age
200	8	59	24.627	63.096	300 GB	10 yrs

4.4 Reengineering and Feature Identification

As discussed in Section 4.2, there are challenges arising from not using a purely platform-based approach. In particular, Table 4.1 shows the scale of the system models and motivates why grouping certain model elements or artifacts together in the form of reusable *features* and explicitly tracing these features can help to manage the complexity of a product family. For example, when propagating changes from one product to another, one can explicitly filter out changes that are not related to common features (see Figure 4.1).

One of the classical approaches to address challenges in the variant management is to develop the product line using a platform-based approach (e.g., SUPERMOD [290]). The literature proposes several solutions to reengineer a product line from a given collection of products. In these section, we show how the Minimum Description Length principle can be applied to reverse engineer a product line from existing products. While we will also discuss the limitations of this approach—and probably all reverse engineering approaches in large-scale industrial settings as the one presented in this chapter—the main purpose of this chapter is to present another application of the Minimum Description Length principle in the context of (model-based) software product lines. Since reengineering of software product lines is not a core part of this thesis, we will not go into any details in this section.

Reengineering of a software product line from a set of products is a challenging task as it requires the identification of commonalities and variabilities among a set of related software artifacts. Reengineering of a software product lines typically involves three phases: (1) feature detection (i.e., feature identification and feature location), (2) analysis (i.e., derivation of a feature model encoding the commonalities and variability), and (3) transformation, that is, using a mechanism to build the products from the features and a configuration (i.e., a feature selection) [22]. In some cases, features have not yet been defined and as part of a domain analysis, one might want some (automatic) support to identify features from a set of related products.

4.4.1 Reengineering a Product Line: Related Work

Several approaches to reengineering software product lines from a collection of software products have been proposed in the literature [2, 22, 207, 214, 276, 277, 293, 297]. To identify features from a collection of products, in principle, one has to identify commonalities and differences of the given products. If some synchronization mechanism has been used to maintain the products, one might hope for identifying features candidates by a structural analysis of the products.

For example, Ryssel et al. [276] propose various clustering techniques to build clusters of subsystems to identify hierarchical features. Alternatively, Formal Concept Analysis (FCA) has been proposed to identify features from a set of products [293]. The idea of these approaches is that model or source code elements (e.g., packages, classes, methods) that appear together in some products but not all the products might constitute initial candidates for features. Often, these feature candidates will then be post-processed, for example, by dividing them into coherent sets of elements and removing noise via techniques such as singular value decomposition [293]. Also, some approaches rely on techniques from the field of natural language processing in order to match different but probably semantically similar elements [22, 293].

Another significant contribution called BUT4REUSE by Martinez et al. [214] offers a unifying framework for adopting software product lines in a bottom-up manner. Their tool alleviates issues like abstraction lack and requirements consolidation across different objectives of software product line adoption by utilizing intermediate models that adapt distinct software artifacts into recoverable elements. It supports various artifact adapters, such as those for EMF models, to facilitate systematic reengineering efforts across a diverse range of software artifacts.

Fischer et al. [99] propose another bottom-up approach called ECCO (Extraction And Composition For Clone-And-Own), which enhances the clone-and-own paradigm by systematically supporting the reuse of artifacts—supporting different kinds of artifacts via an adapter mechanism—to foster new product variants.

Given a set of feature candidates, approaches based on Formal Concept Analysis [53, 277] or logic-based techniques [297] (e.g., discovery of prime implicants in propositional formulas) have been proposed to synthesize a feature model that describes the relationships and constraints between the feature candidates.

4.4.2 Reverse Engineering a Product Line via Minimum Description Length Principle

For this section, we will make some simplifying assumptions.

Definition 4.4.1 Product, Product Line, Feature (Simplified).
A product P is a set of (model) elements $P := \{e_1, \ldots, e_n\}$.
A product line $\mathcal{M}$ is a set of products $\mathcal{M} := \{P_1, \ldots, P_m\}$.
A feature F is a set of elements (and other features) $F := \{e_{F_1}, \ldots, e_{F_l}\}$.

Similar to the arguments in Section 3.5.4, a reasonable set of features would be one that allows for a minimal construction of the set of products. Figure 4.3 shows a

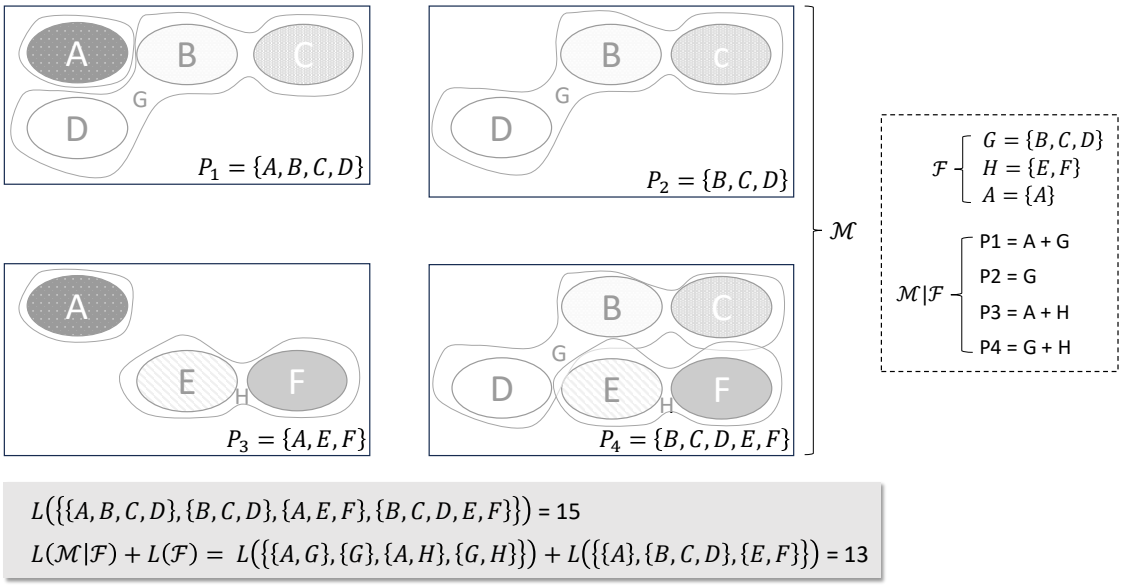

Figure 4.3: Schematic example of Minimum Description Length for the derivation of features. Four products $\mathcal{M} := \{P_1 \ldots P_4\}$, consisting of a selection of elements $\{A, B, C, D, E, F\}$. Counting the description length of a single element (and also feature candidates) as 1 (i.e., we are not using a proper minimal prefix code for their representation for the sake of simplicity of this example), and also ignoring a description length for the brackets (delimiting distinct features and distinct products), we have a description length of 15, if we are not using any features, and 13 if we define the features $\mathcal{F} := \{\{A\}, \{B, C, D\}, \{E, F\}\}$.

schematic example of how the Minimum Description Length principle can be applied to derive features from a set of products.

Note that in this formulation, features can be nested, that is, a feature can contain other features. Similar feature candidates would be obtained by utilizing attribute and object concepts in formal concept analysis. The feature candidates identified by an approach like this are typically coarse-grained (as not all products that can be build from the features are necessarily present in the product line). Furthermore, there is not yet a distinction between features and feature interactions: Elements might be the result of two or more features interacting with each other, for example,

if a feature for monitoring and a feature for persistence are added, there might also be a monitoring of the persistence feature, which requires additional elements.

Based on similarity measures (structural similarity or semantic similarity), one might want to further refine the features [293]. Also, further heuristics can help to identify feature interactions. For example, if, after refinement, a feature candidate is only present if two other candidates are present, but not if only one of them is present, this might be an indication for a feature interaction.

Again, deriving a feature model from a set of feature candidates and given configurations can be described as a model selection problem and can be solved using the Minimum Description Length principle. In practice, existing solutions [53, 296, 297] can be considered as concrete solution approaches to this minimization problem. A brute force approach to this problem would be to consider all possible feature models and configurations and select the one that minimizes the description length, which is infeasible for most (real-world) product lines. Anyway, we will not go into any details of existing approaches and refer the readers to the excellent existing descriptions [53, 276, 293, 296], as the purpose of this section is to show an application of the Minimum Description Length principle in the context of (model-based) software product lines, on the one hand, and given an example of the extent of complexity of real-world product lines, on the other hand.

4.4.3 Feature Mining in a Real-World Setting

In order to discover the merits of feature extraction approaches in the real-world setting described in this chapter, we applied a feature identification approach based on Formal Concept Analysis (FCA), implemented in the BUT4REUSE tool [214]. In particular, the approach investigated in this chapter is restricted to a purely structural analysis. We only shortly describe the experiments we conducted and refer the reader to the Bachelor's thesis by Lukas Selvaggio [291] for further details.

Research Questions for Experimental Context

The purpose of this case study was to evaluate the feasibility of feature identification in a large-scale industrial setting. The approach that we study here is certainly not the best approach available (e.g., we refrain from using any sophisticated post-processing, and we rely purely on a structural analysis). Still, the application to a real-world setting can shed some light on limitations of contemporary approaches, in general. To this end, we want to answer the following questions:

RQ 1: *To what extent do domain experts agree with the identified feature candidate?*

RQ 2: *What are limitations of a structural analysis for feature identification?*

Operationalization

Data: As described earlier, the original models are stored in a proprietary version control system for the modeling tool MAGICDRAW used by the engineers. Anyway, there is an export functionality, which allows us to export specific subsystems and revisions as `XMI` (a model interchange format based on `XML`). These exports can then be used in `EMF` ecosystem. For this study, we fix a certain point in time and apply the approach to a snapshot. The product line is furthermore subdivided into several part systems that group modeled software components semantically. For example, there are part systems for sanitary facilities or part systems for the drive and break control of the train. In total, there are 59 part systems available, from which we selected 5 for this study.

Tool Setup: BUT4REUSE features an adapter for handling `EMF`. To identify commonalities and variation across a given set of (`EMF`) products, BUT4REUSE leverages the EMF DiffMerge API. For this end, one needs to define atomic elements, and how they will be matched. In our concrete case, we have identifiers for each element available since they are maintained by MAGICDRAW. We choose the atomic elements to be the `EModelElement`[5] and use their identifiers for the matching.

We generated a formal context and computed a pruned formal concept lattice, defining feature candidates as groups of elements aligned in the same attribute concept.

Study Setup: To evaluate RQ 1, we employed qualitative analyses, including focus groups with a group of domain experts to assess identified features. The focus group consisted of the author of this thesis, a Bachelor's student, two lead architects in the project, with 10 years of job experience and 6 years of experience within the project, and the head of tool development, with 15 years of job experience and 6 years of experience within the project. The feature candidates were presented via a visual highlighting of the elements in a selection of 15 diagrams via a MAGICDRAW plugin that we developed for this purpose (see Figure 4.4).

We then asked the participants about the *completeness*—are all elements that belong together in this diagram also highlighted with the same color—and the *coherency*—are there elements in the highlighting that should be part of another feature—of the highlighted elements. The participants had to rate the completeness and coherency of the highlighted elements on a Likert scale from 1 to 5, where 1 means that the highlighted elements are not complete or coherent at all, and 5 means that the highlighted elements are complete or coherent. As a baseline, we also asked the participants to rate the completeness and coherency of randomly generated feature candidates and manually manipulated feature candidates. Finally, we also discussed their assessment in a group discussion.

5 `EModelElement` is a superinterface of attributes, classes, and references in `EMF`.

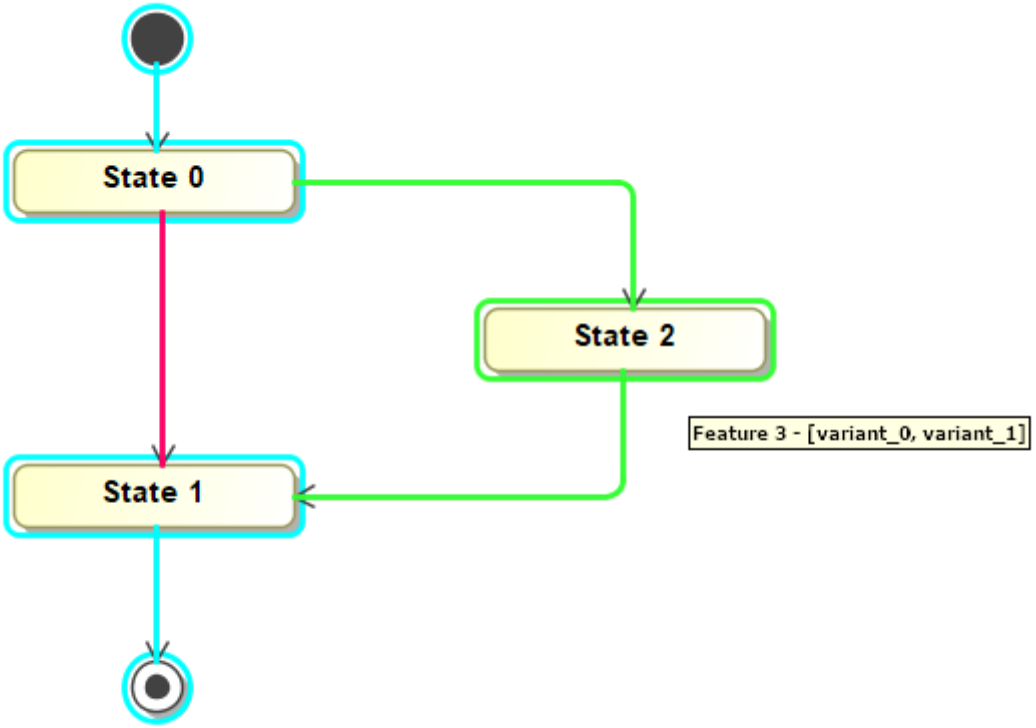

Figure 4.4: Screenshot of the MAGICDRAW plugin that we developed for this study. The plugin allows for highlighting elements that are part of a feature candidate.

To evaluate RQ 2, we qualitatively discuss the results of the feature identification approach and qualitative feedback from the domain experts.

Results
Regarding coherency, we find that the feature candidates identified by the FCA approach are rated significantly higher than the randomly generated feature candidates ($p = 0.047$). The completeness of the feature candidates identified by the FCA approach is not rated significantly higher than the randomly generated feature candidates ($p = 0.594$). The mean values for coherency and completeness are shown in Table 4.2. Our analysis also suggests, that there is a dependence on the diagram type in which the features are highlighted. For example, for `SysML State Machine Diagrams`, the participants rated the coherency and completeness of the feature candidates higher than for `SysML Internal Block Diagrams` or `SysML Block Definition Diagrams`. Anyway, the sample size per diagram type is too small to draw any general conclusions. In an informal group discussion, the participants mentioned that the highlighting of feature candidates in the diagrams indeed provide valuable insights

	Coherency	Completeness
Random	2.0	1.7
Manipulated	1.4	1.5
FCA	3.3*	2.0

* Significance level $p < 0.05$

Table 4.2: Mean values for coherency and completeness of the feature candidates identified by the FCA approach, randomly generated feature candidates, and manually manipulated feature candidates.

into the variability of the products, but they could not see features in the feature candidates.

This brings us to the results of the second research question. For the five part systems that we analyzed, we obtained between 333 and 497 feature candidates, with a mean of 455 feature candidates.

Figure 4.5 shows a concept lattice for one of the five part systems that we analyzed. The concept lattice shows the feature candidates as nodes and the relationships between the feature candidates as edges. On the one hand, some candidates (especially in the common block of model elements and the product-specific blocks of model elements) are very coarse-grained and could probably be further refined. On the other hand, the approach considered here identifies a large amount of fine-grained feature candidates. A closer look into the feature candidates revealed that there is a lot of "noise" in the feature candidates. One of the major sources of noise that we could identify in the focus group is variability smells. For example, a modification that has been conducted in a specific product variant but has not been propagated to other product variants, will lead to a feature candidate that is incomplete. At the same time, the modified feature will also be broken apart and also looses its completeness. Furthermore, it can happen that during variant synchronization, some parts are remodeled in a specific product (e.g., because of issues during the merge process). This also leads to semantically identical elements that are not part of the same feature candidate, because they will have different identifiers in the model. Also, feature interactions will be hard to distinguish from the feature candidates, as the approach considered here does not consider feature interactions.

Discussion

The results of the focus group highlights several limitations of the approach considered. We also see some potential in the approach, although not for feature identification. Indeed, most of the identified feature candidates are not actual features and the products are highly fragmented into the candidates.

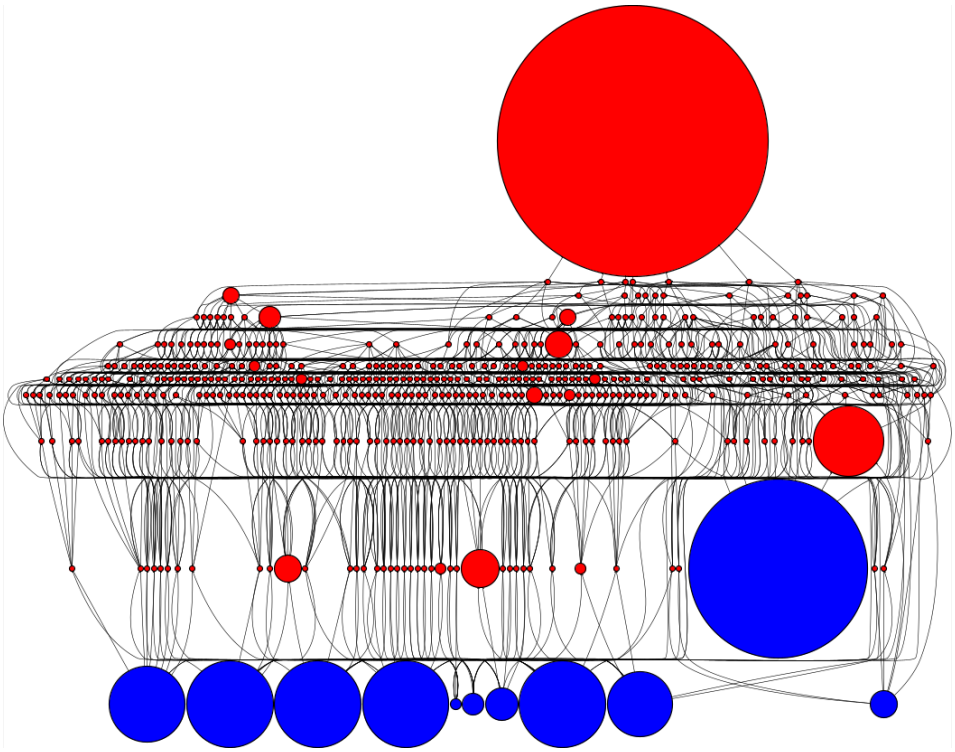

Figure 4.5: Depiction of a concept lattice for one of the five part systems with 11 product variants. The size of the circles corresponds to the log of the number of elements in the feature candidate. In the top of the figure, the common block is clearly visible, accounting for many elements. The blue circles in the bottom of the figure are the product specific model elements. This example demonstrates the complexity of the feature identification problem in a real-world setting.

RQ 1: *To what extent do domain experts agree with the identified feature candidate?* The feature candidates identified by the FCA approach can be considered as rather coherent, but not complete. Indeed, the different product variants are highly fragmented. Some feature candidates (e.g., the common block) are very coarse-grained, while others are very fine-grained and sometimes rather reflect variability smells than actual feature candidates.

The reasons for the high fragmentation of the products are manifold. One of the major origins of the fragmentation is improper or incomplete synchronization of the products. This also suggests that a necessarily not "perfect" semantic matching of elements across product variants can hinder the identification of feature candidates. Given the noise, a distinction between feature candidates and feature interactions is hard to make.

RQ 2: *What are limitations of a structural analysis for feature identification?*
The approach can not effectively handle noise, can not distinguish between feature candidates and feature interactions, can not identify feature candidates that might be present in the product line but have not yet been used independently in one or more of the products. Overall, there are many fine-grained feature candidates that are not actual features, while some coarse-grained feature candidates might include more than a single feature. The approach is severely impacted by variability smells.

The extent of the fragmentation with 455 feature candidates per part system, on average, makes it highly unlikely that—even with some further pre- and post-processing—the approach can be used to identify features in a real-world setting of the complexity of the product line considered here. Most heuristics come with their own limitations (e.g., statistical approaches are almost certainly not 100% correct), adding another level of noise, which decreases the quality of the results of a Formal Concept Analysis. Therefore, in order to identify features in a real-world setting, a more resilient approach is needed, or one needs to remove noise from the models before applying the approach.

As an interesting side note, the participants mentioned that the highlighting of the feature candidates helped them to quickly identify variability smells. Therefore, the approach considered here might be used in a first step to identify and clean-up variability smells, before an extractive feature identification approach is applied to reengineer the product line.[6]

4.5 Semantic Lifting to Reduce the Complexity of Model Differences

While in the previous section we have seen that a migration from managed cloning towards a feature-oriented approach is not feasible in the short term, we now discuss an approach to reduce the complexity of model differences. In this section, we introduce an approach to "semantically" summarize model differences. The details of the approach will be given in Chapter 6.

The basic motivation behind the approach is as follows: As a consequence of unintentional divergence, large model differences (e.g., thousands to hundreds of thousands of changes) can increase the efforts for activities, such as change and

6 Since our initial study in summer 2022, the team has decided to iteratively trace model elements (`Software Components`, in particular) to features that have been manually identified by the domain experts. We furthermore conducted a study to investigate the merits of the highlighting to support the identification of variability smells in the product line, and we found that the approach is indeed helpful to identify smells in the models related to variability issues.

feature propagation, quality assurance, or domain analysis. As Figure 4.1 indicates, the differences in many of the activities are inflated by unrelated differences. Other than common variability management approaches, we are therefore also interested in the feasibility of a treatment of the large model differences, that is, a treatment of the symptoms of unintentional divergence.

In this section, we describe an approach to reduce the complexity of model differences that is based on *semantic lifting* [163] and edit operation mining [324], which we will discuss in greater depth in Chapter 6. Semantic lifting is not a product line engineering approach but a method for aggregating model differences, in general.

4.5.1 Description of the Approach

The high-level idea of the approach is to simplify the presentation of model differences and to provide further insights for the user. This can speed up tasks such as change propagation. We approach the problem of large model differences by reducing the "perceived" size of the model difference. This is realized by grouping fine-grained changes to higher-level (semantic) changes. These changes can also be classified or tagged (e.g., a change can be a violation) to support the architect or developer during analysis and quality assurance related tasks. Furthermore, higher-level changes can help in the identification of common functionalities.

Technically, we achieve this by a semantic lifting of the model differences. As shown by Kehrer et al. [163], a set of defined edit operations (or high-level changes, in our case) can be recognized in a model difference, and each model difference can be expressed in terms of these high-level changes. In this semantic lifting approach, high-level changes are defined by *edit rules*. These edit rules are an executable description of the edit operations (templates) defined in Section 2.4.2. Edit rules are then "lifted" to recognition rules, which can be applied to a model difference to group the low-level changes in the model difference into *semantic change sets*. Furthermore, if low-level changes are contained in more than one change set, a heuristic is applied to select only disjoint change sets. Larger change sets are preferred over smaller ones. The result of lifting is a difference containing all original changes grouped into disjoint semantic change sets. For example, a change pattern that we observe quite often in the models from our case study in Section 4.3 is a connector and two ports added between two existing parts in an SysML internal block diagram (see Figure 4.6). This high-level change comprises 17 low-level changes. For many tasks related to the model differences, it will be sufficient to see this high-level information and only drill down if necessary.

In a subsequent step, filters can be applied to the lifted differences, which is especially helpful for quality assurance. Some changes will be less interesting, while others will be more interesting. For example, a renaming operation might be less interesting compared to adding interfaces between components. Filtering these kinds of changes improves the clarity of the model difference, especially since these changes

Figure 4.6: Results of the semantic lifting of a connector that is added between a component and a subsystem.

will typically be scattered across the model difference. Furthermore, illegal changes can be highlighted in the model differences by tagging certain edit operations.

A difficulty of this approach is that the high-level changes have to be known and defined as input to the semantic lifting step. This requires a deep knowledge of the modeling language's meta-model (i.e., SysML, in our case) and the underlying paradigm of the transformation language. Some high-level changes might even be project-specific. Moreover, some changes are a form of tacit knowledge [260], and it will be hard for domain experts to externalize this knowledge. Leaving the definition of high-level changes to the engineers of the project causes a large overhead, which we want to avoid. For this reason, we developed a recommender system, OCKHAM, which is able to mine high-level change patterns from model repositories [324]. We will present OCKHAM in Chapter 6. The tool is based on the Minimum Description Length Principle [118], discussed in Section 3.5. OCKHAM considers the problem of automatically identifying semantic change sets from a graph mining perspective.

We will see in Chapter 6, using the models from the case study presented here, that the automatically discovered change patterns are relevant and help in a meaningful manner.

Compared to the original semantic lifting approach [163], our combined approach *automatically* recommends meaningful edit operations for lifting model differences.

4.5.2 Putting the Pieces Together: Implementation

The engineering team at Siemens Mobility uses MAGICDRAW for their SysML models, but there is an export to EMF. Many tools in the project, such as code generation tools, are based on EMF. The approach presented in Chapter 6 builds on top of an "intermediate" EMF model. For semantic lifting, we use SILIFT [163], which uses SIDIFF [284] for computing structural differences. For the definition and execution of the recognition rules, SILIFT uses HENSHIN [19]. SILIFT outputs a list of changes and semantic change sets (see Figure 4.6), which can then be filtered and sorted. For the edit operation recommendation, we use OCKHAM [324], which uses SIDIFF to compute model differences and GASTON [242] for frequent subgraph mining. OCKHAM outputs the recommended edit operations as HENSHIN rules.

4.6 Empirical Analysis of The Case Study

To better understand the challenges faced in the concrete real-world project, we talked to practitioners in the project. We analyze the symptoms of model drift, influencing factors, and possible solutions.

4.6.1 Research Questions for Experimental Context

In the study reported in Section 4.4, we already found that a migration to feature-oriented approach can be challenging, that is, the effort and cost of defining and tracing features can be high. Here, we focus on the challenges of model drift in the managed cloning approach at Siemens Mobility and possible alleviation. To better understand the model drift, we set out to answer the following research questions. RQ 1 – RQ 3 study the problem side, while RQ 4 and RQ 5 study the solution side:

RQ 1: *Can model drift be observed in this product line, and how is it perceived by the developers?*
RQ 2: *What are symptoms of model drift?*
RQ 3: *What are causes of model drift and large model differences?*
RQ 4: *Can solutions proposed in the literature (i.e., bottom-up approaches and platform-based approaches) be applied to reduce model drift?*
RQ 5: *Can the semantic lifting approach support the engineers in handling large model differences?*

4.6.2 Study Setup and Conduct

To answer our research question, we use a mixed methods study [13, 85].

Table 4.3: Overview of the interviewees.

Role	Scope	Experience (level)
System engineer	Single model	Entry-level
System engineer	Single model	Intermediate
Tool support	Cross-cutting	Intermediate
Chief architect	Cross-cutting	Senior
Head of tool development	Cross-cutting	Senior

Semi-structured interviews
To get a first impression of the model drift challenge in this project, we conducted semi-structured interviews with five engineers of the project (see Table 4.3), among them the head of the tool development team and one of the lead architects. Three of the participants were working with more than one of the products of the product line and two of the participants did contribute to a single product. The interviews lasted 45 – 60 minutes. The interview had three sections: Challenges in clone-and-own product line engineering, unintentional divergence, and variability management solutions. In the last section, we presented the approach from Section 4.5 and asked the participants for their feedback regarding technical, organizational, as well as economic feasibility. We asked only open-ended questions and did not have any further constraints except covering the three sections mentioned above.

Questionnaire Survey
Based on the results of the semi-structured interview, we designed a questionnaire. We conducted a web-based[7] questionnaire survey with 9 project employees of the project (cf. Section 4.3). The questionnaire was distributed via a mailing list of the architects of the project. Therefore, the majority of the recipients were architects and developers involved in architectural decisions. The survey had four sections; the first section asked the interviewees about their role and experience, the second about evolution and trends of the project, the third about model differences in the project, and the last one about solution ideas and their feasibility. We gave an introduction to the topic and shortly summarized existing variability management solutions (cf. Section 4.2.2). We also explained the semantic-lifting approach. All together, we asked 27 questions. Since a majority of the employees are German speaking, we asked all the questions in German. The questionnaire was sent to 17 candidates and 9 filled out the questionnaire. 5 of the respondents were architects, while 4 of the respondents identified themselves as developers. The project experience of

7 We used Microsoft Forms, since it is easily accessible and employees at Siemens Mobility are familiar with this tool.

the respondents ranged from 1–2 years to more than 6 years. The questionnaire is available at https://forms.office.com/r/WRvUNQisuR.

Repository data
To also get quantitative insights, we used the SysML models in the project stored in a the version control system TEAMWORK SERVER, which, at the time of the survey, was the default solution for collaborative development in MAGICDRAW. There is one project for each of the submodels (also called part systems) and the repository holds the entire history for each of the submodels and all trainset platforms as well as the common platform. In Chapter 5 details of the dataset (i.e., the INDUSTRY dataset) are given.

To answer the research questions, we use the input from the semi-structured interviews, the results from the questionnaire, as well as insights from the models in the model repository.

4.6.3 Results

Semi-structured interviews
During the interviews, four out of five interview partners mentioned challenges related to large model differences when asking about general challenges in their project. For example, they stated that during the propagation of changes from one product variant to another variant, the model differences can be hard to understand because of changes concurrently made by other developers. One of the interview partners did not mention challenges related to the variability at all. When we asked about the tasks and roles in the project, it turned out that she was mainly involved in the evolution of a single product and not in any activities related to variability management. As major drivers for variability, differences in hardware as well as different customer requirements have been mentioned.

All interviewees ruled out a migration to a platform-based product line engineering approach, because of migration efforts, risks, and immature tooling. Furthermore, there are less than a dozen of product variants, and therefore the effort for maintaining the platform possibly would exceed the benefits. Bottom-up approaches are conceptually more realistic, but there exists no mature tooling yet and, as a prerequisite, the variability has to be known explicitly. Making variability explicit (e.g., in the form of feature traces), has been defined as a mid-term goal, but not something feasible in the short term (cf. Section 4.4).

The most time-consuming tasks for the interviewees are variant synchronization tasks (e.g., change propagation). Merging appears to be difficult because it is hard to define what belongs to a feature and what does not. There are many interfaces between different submodels, which further complicates merging. Computing the differences themselves is already time-consuming and takes 15 to 30 minutes per

model difference. Worse, many fine-grained changes, such as renaming operations, make the model differences hard to read. This is also the case for frequently performed refactorings that are introduced because the models are adapted to new tooling. When we explained semantic lifting to the interviewees, four out of five participants stated that a semantic lifting approach has the potential to speed up several of their activities in the project. For example, the analysis of model differences for review purposes could be supported, and irrelevant changes can be filtered out.

Questionnaire
All the answers were optional, but most of the 9 participants of Siemens Mobility answered to all of the closed-ended questions. We also received 10 answers to open-ended questions.

78% of the respondents said they observe a drift between the different products. All of them see model drift even among the 5 biggest challenges in the project. 67% of the respondents say that, without a solution, serious problems might arise; 22% of the respondents say that a solution will increase the efficiency.

As symptoms of model drift, respondents report increasingly large model differences (78%), increased effort for change propagation (67%), increasing numbers of inconsistencies (56%), increase in communication efforts (44%), increase in feature interaction bugs (22%), and increased effort for domain analysis (33%). Furthermore, for 89% of the respondents, model differences play a role in some of their tasks in the project, for 33% even a significant role. While for only 22% of the survey participants model differences *between two revisions* within one product variant can be challenging, 78% of the respondents experience model differences between two product variants as challenging. For these respondents, differences play a role and can be challenging during the following tasks: reviews (89%), changes and bugfixes *between two variants*(100%), and domain analysis (89%). One of the participants pointed out that, even in requirements engineering (requirements are also part of models), unintended divergence is challenging.

As causes of the model drift, all the respondents mentioned different hardware of the products and parts of the architecture not being reused, 33% even report a significant influence of not reusing parts of the architecture. 67% think that it is difficult to define and unify features across multiple products and evolution of the single products has been mentioned by 44%. According to the respondents, large model differences are also caused by the size of the models themselves (78%), missing explicit variability information (67%), missing tool support for filtering and cherry-picking of model differences (100%), and no distinction between architectural (i.e., refactorings) and functional evolution (78%).

In Figure 4.7, we show a violin chart of the Likert scale data for the survey participants' opinions about 150% platform approaches, bottom-up approaches, and the semantic lifting approach. The plots show their opinions along several dimensions: whether the approaches are suitable to provide improvement regarding the model

drift challenge, whether the advantages of utilizing these approaches would outweigh the cost and risk of development and migration, whether the current tooling would support the approaches, whether the approaches are organizationally and technically feasible, and whether tooling for these approaches is known already by the practitioners.

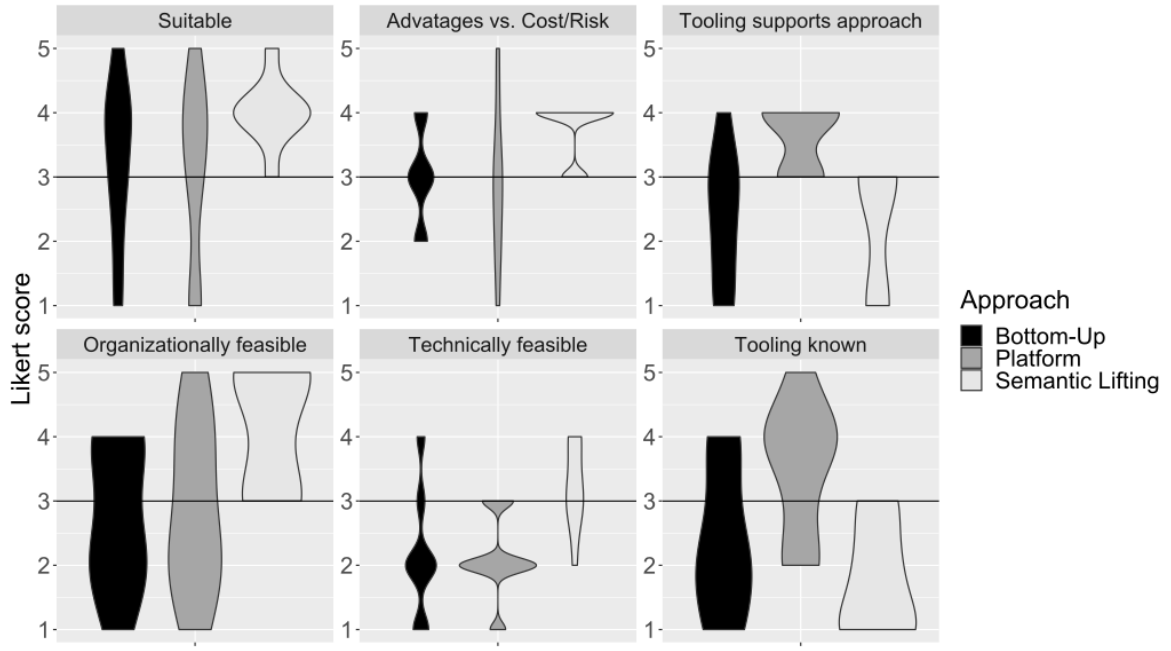

Figure 4.7: Comparison (violin plots) of the opinions about bottom-up approaches, platform approaches, and the semantic lifting approach. Likert scores range from 1 (=Disagree) to 5 (=Agree).

Repository data

In order to also quantify the model drift, we computed the model differences between the common platform and the trainset platform for a selection of 10 submodels and trainset platforms and three time regions (we can not take time points, because we do not have revisions at the same time points for all of the submodels). Figure 4.8 shows a violin plot of the size of the model differences. In this plot, we grouped the differences into three groups (Spring 2022, Spring 2021, Winter 2019). Even more, if we consider a single trainset platform and a single submodel, we can observe an increase in the size of the model difference.

We also made specific observations in the model histories, specifically with the purpose of cross-validating some of the statements of the interview and survey participants. For example, we can see that there are entire modules with the same purpose that are not reused but evolve independently for each of the trainset platforms. We also see that refactorings, such as renamings or changing the structure in the package hierarchy can obfuscate model differences. For example, the renaming of

a root package will lead to a change in the *fully qualified name* of all subpackages. Also we encountered revisions of several part system for which computing the model differences using MAGICDRAW failed because of a memory limit. We will refer to specific observations like these directly in the discussion in Section 4.6.4.

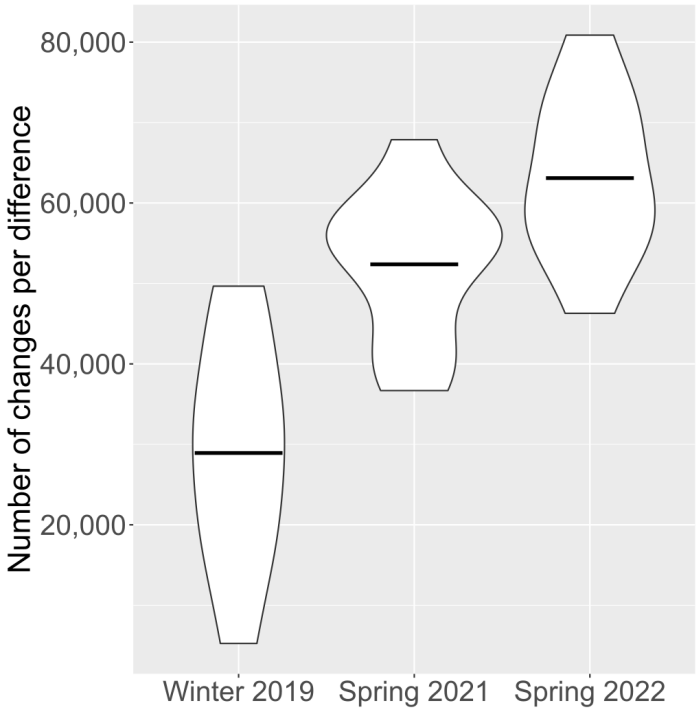

Figure 4.8: A violin plot showing the increasing number of changes per difference between common platform and trainset platforms.

4.6.4 Discussion

In this subsection, we use the results from the experiments to answer the research questions from Section 4.6.1.

RQ 1: Model drift. Four out of five participants of the semi-structured interviews as well as 78% of the survey participants stated that they experience model drift and that model drift is a serious challenge. As can be seen from Figure 4.8, in terms of the size of model differences, the common platform and the trainset platform experience a drift. The time for computing the differences of the latest models in MagicDraw was 25min, on average, on a Lenovo P51 with an Intel Core™ i7-7820HQ CPU and 10 GB RAM for the MagicDraw Java Application.

RQ 1: *Can model drift be observed in this product line, and how is it perceived by the developers?*
Model drift is perceived as serious challenge for the presented case study. This drift is manifested in the model histories and is also perceived by a majority of the engineers from our interviews and questionnaire.

RQ 2: Symptoms of model drift. According to the interviews as well as the questionnaire, change propagation, domain analysis, and quality assurance are tasks that have become increasingly challenging because of the model drift. This is mainly because of the huge size of model differences involved in these activities. For example, for the heating, ventilation, and air conditioning part system (i.e., one of the 59 submodels), MAGICDRAW presents 81.000 differences between one trainset platform and the common platform. This induces considerable effort for analysis tasks and variant synchronization tasks (e.g., change propagation). For instance, in our interviews, the engineers at Siemens Mobility confirm that, on average, the analysis for quality assurance purposes takes 30 minutes per submodel. Given that often multiple submodels have to be analyzed on a regular basis—often several times per month—this leads to significant efforts and costs. Furthermore, the risk of making mistakes, for example, missing a touched *sealed component* (i.e., a component which is marked as unmodifiable), raises. If disallowed changes are discovered only late in the process—at worst during software qualification—this leads to delays in the development process which, in the end, leads to delayed delivery of the products to customers and penalties by customers. Another symptom of model drift are increased communication efforts, which were reported by 44% of the participants.

Missing reuse opportunities—due to the sheer number of differences—further accelerates the drift between the different trainset platforms. This causes redundant work for the different trainset platforms. According to one of the survey participants, "rebasing of a product to a new version of the parent platform is not possible for months because the diffs are so large that tool crashes reproducible," which then leads to work been done multiple times—sometimes even new duplicates being created.

Besides the sheer number of changes, these changes may be scattered across the entire model difference. As an extreme example, recently the engineering language has been changed from German to English. In total, more than 7.5% (or 98,486 changes in total) of the changes are related to renaming refactorings (to a large extent due to the change of the engineering language). For some analysis tasks (e.g., reviews), these changes are irrelevant, but the modeling tool does not provide filters for these changes. For other tasks, such as domain engineering, it would be helpful, to classify these changes and merge them contiguously instead of being intermixed with other types of changes (e.g., functional changes).

RQ 2: *What are symptoms of model drift?*
Large model differences and increased communication efforts are the main symptoms for model drift experienced at Siemens Mobility. Finding commonalities or reuse opportunities in these large differences has also become challenging. Mostly the three tasks change propagation, domain analysis, and quality assurance seem to be (negatively) affected by the drift.

RQ 3: Causes of model drift and large model differences. We found the following drivers for model drift and large model differences.

Huge models: According to the interviewees, one reason for the enormous size of model differences is the *size of the models* themselves. The models grew steadily in size over the last seven years. The average size is 24.627 objects, where an object is an instance of a class in the meta-model. That is, attributes and references are not even included in this count. The reason is that the train software and system is modeled with a high degree of detail. Various additional information is provided, for example, requirements traceability or organizational information.

Duplicated architecture: In the model repositories, we found parts of the models that seem to have a similar purpose but are modeled individually for every trainset platform. These duplicated parts appear in every model difference and also inflate the number of changes in these differences. Different hardware used for the different trainset platforms appears to be one of the main drivers for inducing new variability and duplicated parts of the models. The interviews and survey support this finding. One of the survey respondents said that "under time pressure, often only a single solution is sought for the concrete problem in the concrete project instead of a solution suitable for the platform."

Implicit variability: Many of the problematic changes that are typically found during quality assurance are related to missing explicit constraints related to variability. For example, there are components which must not be modified in one of the trainset platforms. Because this information is often implicit, trainset platform engineers sometimes change parts of a model where actually no change should occur. Missing variability information increases the size of the model differences: Features that are only part of the platform but not present in a specific train type will be always shown in a model difference, even though they might not be relevant. Likewise, features that are only part of the specific trainset platforms but irrelevant to the platform should be excluded from the platform when merging train type changes. For tasks such as extracting commonalities, such features are of special interest. This extraction would be easier if only product-specific features are compared across the different train types, that is, explicit variability should be used for isolated feature development.

Missing tool support: Especially for domain analysis, a model difference between a trainset platform and the common platform or between two trainset platforms is computed. In the differences, there will be many changes that are not relevant for the

domain analysis. One limitation of MAGICDRAW is that it does not support filtering model differences. Even given a concrete feature mapping, MAGICDRAW does not provide any filtering functionality out-of-the box. This is also partly related to the fact that there is no commonly accepted way to express variability in SysML models. The engineers at Siemens Mobility not only need support for Boolean features but also numeric and topological features. Furthermore, MAGICDRAW only supports two-way and three-way merging to propagate the changes between branches. Merging aims at including as many changes as possible, while in different variants, some differences are desired [167]. *Cherry-picking* changes (i.e., only selecting changes between two consecutive revisions) would support change propagation, but MAGICDRAW does not support cherry-picking as of now. One of the survey respondents mentioned that "poor tool support for taking over simple changes leads to the same changes being brought in individually and thus only being the same, but not the same elements, so that technically further diffs arise." In our repository data, we can also see these duplicated blocks, which will show up in the model differences.

Classification of changes: Model differences typically show many fine-grained changes. Some of these fine-grained changes belong to higher-level changes, but they are not grouped together in the presentation of model differences. This increases the "perceived" size of the model differences. Another challenge that increases the complexity of model differences is the lack of a distinction between architectural evolution (i.e., refactoring) and functional evolution. For example, correcting a typo in a component name is presented in a similar way to the user as changing a connector. Another example are move refactorings, for example, a package is moved in the containment hierarchy, or graphical changes in diagrams. As one of the survey respondents states, "[...] semi-automatic changes (example: upper/lower case of elements) lead to HUGE diffs, so that no manual diff (and merge) is possible for a long time." These changes are typically scattered across a model, and they are ubiquitous. For some activities, such as quality assurance, it would be helpful to distinguish these changes from other, more relevant changes.

RQ 3: *What are causes of model drift and large model differences?*
Large models together with a lack of support for explicit variability as well as missing tool support for filtering model differences result in large numbers of model differences that have to be analyzed. Therefore, it is increasingly difficult to reuse parts of the system across multiple trainset type platforms. Parts of the models can not be reused even though they have a similar purpose. These factors cause the variants drifting apart over time.

RQ 4: Solution approaches. During our interviews and in the survey, we identified several hindrances using existing solution approaches (see Section 4.2.2), which we discuss in this subsection w.r.t. possible solution approaches.

Platform approaches: Our interviewees and survey participants mentioned several barriers to use 150% platform approaches. Most prominently, platform approaches can not be used because of the cost and risk of a migration (cf. Section 4.4). At Siemens Mobility, there is already a significant amount of modeling data that would have to be migrated. One of the participants of the interview mentioned that "many of the developers think in terms of concrete products and not the entire product line". This also shows that a migration from managed cloning to a (150% platform) software product line approach will face organizational hindrances. Furthermore, there is no variability modeling approach that supports Boolean constraints, numeric constraints, as well as also topological constraints. One of the survey participants even said that "the structural variability is so large that it can not be modeled using a 150% platform." Some mentioned that such an approach could be useful as a basis (which is also a goal of current efforts) but some of the differences will be too special and need individual solutions.

Bottom-up approaches: According to our interviews, a support for understanding the variability and commonalities between the different trainset platforms is clearly desirable. Nevertheless, according to interviews and survey, approaches proposed in the literature are not mature enough. For example, as for the 150% platform approaches, there is no approach that supports Boolean constraints, numeric constraints, and topological constraints. Additionally, variation control systems proposed for models (e.g., SuperMod [290]) have not yet been evaluated in an industrial setting. This is complicated by the fact that there is no universally accepted standard for expressing variability in modeling languages [34].

Typically, bottom-up approaches make use of traces between features and implementation artifacts. Automatically reverse engineering the feature traces in this product line is not feasible (cf. Section 4.4). In general, the "development of methodology and tooling in the domain would be of enormous complexity", according to a survey respondent. Nevertheless, the engineering team at Siemens Mobility developed an approach to feature traceability, and the manual incorporation of explicit variability information into the models is an ongoing activity. Still, there are only a few feature traces yet which could be used.

RQ 4: *Can solutions proposed in the literature (i.e., bottom-up approaches and platform-based approaches) be applied to reduce model drift?*
Common variability management solutions proposed in the literature are not applicable to the setting at Siemens Mobility. This is mainly due to immaturity of the available tooling, but also because many approaches make simplifying assumptions, not taking real-world and organizational constraints into account. A migration to another approach comes with high costs and risk, in any case.

RQ 5: Semantic lifting. While platform-based and bottom-up approaches are not (yet) applicable in this setting, the results from the interviews and the questionnaire

survey suggest that the semantic lifting approach from Section 4.5, which is based on our work in Chapter 6, appears to be suitable to support the engineers in handling large model differences.

Having a *semantic change log* for model differences "has already been hypothesized as a valuable solution in a workshop", according to one of the interview participants. Four out of five interviewees stated that using higher-level change patterns in the presentation of model differences would support many tasks, such as merging features, filtering the differences during reviews, identifying illegal changes in model differences, and understanding model differences in general. The remaining participant did not perform any tasks where the current size of the model difference was considered as a problem. Two interviewees, who were responsible for propagating changes between the common platform and trainset type platforms described the merge process as one of their biggest challenge: since they "[...] work with several workers in the project, it is also very time-consuming [during merge] to filter out what my own changes were. [...] Usually several points in the model are affected by a small change—external interfaces, software, product structure, physical structures—so there are changed elements in every folder, so to speak." Grouping related changes can support here as well. Two of the interviewees mentioned that change patterns could also be used to easier recognize complex refactorings in model differences. We show in Chapter 6, that in this product line, recommended edit operations for the semantic lifting achieve a compression ratio of about 4.0, on average (without filtering yet) [323].

The current tooling does not support such an approach natively but, according to one of the interview participants, it would be possible to develop a plugin for MAGICDRAW. Unlike the bottom-up approaches, semantic lifting appears to be more feasible from a technical viewpoint (see Figure 4.7). Nevertheless, one of the survey participants is also concerned that the "[...] current lack of experience in that field, could be an obstacle in the development of a production-ready solution."

RQ 5: *Can the semantic lifting approach support the engineers in handling large model differences?*
Interview and survey participants at Siemens Mobility largely support the hypothesis that a semantic lifting approach mitigates challenges arising from the model drift. The hypothesis is further supported by the observation that the differences can indeed be compressed using semantic lifting in this product line. Nevertheless, there are concerns that the production-ready implementation of a semantic lifting approach can be challenging.

4.6.5 Threats to Validity

We reported on a concrete large-scale product line in an industrial setting. We do not claim the generalizability to other large-scale product lines here, although we strongly believe that similar challenges exist in many industrial settings. This is also suggested by the work of Dubinsky et al. [86] and Rubin et al. [274].

A threat to internal validity is caused by our setup of the interviews and the questionnaire. We had no control group, because we only had a small group of participants. Nevertheless, in particular regarding the existence of the drift, we used three different methods (interviews, survey, and repository data) to validate the presences of an unintentional divergence. Furthermore, we did not evaluate the semantic lifting approach in a long-term application at Siemens Mobility. For future work, we need to put the approach to the acid test by letting the engineers work with the tooling.

A threat to external validity is that we only used a small sample of engineers from the project for our interviews and in the survey. Nevertheless, the engineers participating in the interviews and the survey were involved into architectural decisions or had an otherwise leading role. We also used only a subset of all submodels in the project for our repository analysis. Anyhow, because the results of interviews, survey, and the repository analysis are consistent, we are confident that our results hold for the entire project. Also, many results were evident from a small sample set, for example, we could find the drift for all single products in the sample.

4.7 Conclusion and Outlook

In this chapter—to motivate our research on intelligent model evolution support—we reported on a concrete industrial large-scale model-driven managed cloning software product line. We first described the project setup and the challenges faced by the engineering team at Siemens Mobility. We analyzed the feasibility of a reengineering of the product line and found that the complexity of the models and the presence of noise make an automated feature mining approach challenging. Furthermore, we conducted semi-structured interviews, a questionnaire survey, and we analyzed the model repository to understand model drift in this product line.

We found substantial model drift that impairs the evolution of the product family tasks, in particular, involving tasks such as change propagation, domain analysis, and quality assurance. The main reason is that these tasks are based on large model differences (across products) that have to be analyzed manually by engineers. We found that solutions described in the literature are often not applicable in a real-world setting, for example, filtering model differences for certain features requires a rigorous modeling of variability in the models and existing tools miss some critical functionality, for example, handling structural variability. Instead of addressing vari-

ability management as one of the root causes for large model differences, we also studied the alternative route of addressing large model differences directly. Based on our observations, we proposed a concrete solution to mitigate the problem by summarizing the model differences in terms of higher-level change patterns that are automatically derived from model repositories based on the idea that meaningful patterns are the ones that compresses model differences. In the context of the project at Siemens Mobility, this idea appears to hold true. This approach was also the original motivation to investigate edit operation mining presented in Chapter 6. Our mixed methods study provides evidence that semantic lifting can support engineers in tackling large model differences. The approach is orthogonal to variability management solutions. For example, even using variability management solutions, model differences can still be large (e.g., because of the size of the models or missing tool support for filtering model differences properly).

Many existing solutions to support software product-line engineering related tasks require explicit variability in the models. Establishing a bottom-up, managed cloning product line engineering is a long-term target at Siemens Mobility. The engineering team at Siemens Mobility is currently working on making variability explicit by creating explicit mappings from features to model elements. As a long-term solution, it is conceivable to use this variability information, for instance, to propagate changes between trainset type platforms or for filtering of models and model differences.

While in the long-run it is imperative to establish a working variability management approach, in the short-term, treating symptoms of unintentional divergence can support engineers and avoids costs and risk of more costly solutions.

Also, the feature mining plugin that we presented to the architects and engineers, even though not helpful for feature mining, has shown to provide benefits during quality assurance of the models, since the approach helps to make variability more explicit in the diagrams. In another study [322], we showed that, by loading (parts of) the models into a graph database, many insights can be obtained through simply querying the database. In general, there are huge amounts of data that can be leveraged to support engineers during several activities, as we will show in the upcoming chapters. We will also use models from this case study for our investigations in the upcoming chapters.

5

Datasets: A Balance of Synthetic Control and Real-World Relevance

Errors using inadequate data are much less than those using no data at all.
— Charles Babbage

In this chapter, we describe the datasets used throughout the rest of this thesis. There are three datasets: INDUSTRY—A real-world dataset from the project described in detail in Chapter 4, REPAIRVISION—a real-world dataset of models from the Eclipse Modeling Framework ecosystem, including a project modeling UML2 itself, and SYNTHETIC—a set of synthetic models conforming to an ECORE meta-model. All datasets include histories in the form of revisions of models and therefore constitute a kind of ground truth for the study of software model evolution.

5.1 Datasets Overview

In this section, we give an overview of the three datasets that we use to answer our research questions in the following chapters. Ranching from synthetic, simulated model repositories to real-world industrial repositories allows us balancing internal and external validity. Basic statistics about the datasets are given in Table 5.1. We will also describe each dataset in detail.

5.1.1 Industry Dataset

We created the INDUSTRY dataset from the repository (Teamwork Server) of SYSML models in MAGICDRAW from the real-world product line for train control software described in Chapter 4. The train control software comprises several part systems (also called submodules), corresponding to semantically coherent parts of the train

Table 5.1: Figures for the datasets. Model size only reflect model elements (i.e., not the number of attributes). The number of changes includes attribute changes.

Dataset	Number of Models	Number of Revisions	Avg. Model Size	Avg. Number of Changes	Public Availability
INDUSTRY	59 (and 8 variants)	48,121	24,627	3,300[a]	No
REPAIRVISION	20	3,139	685	69.9	Yes
SYNTHETIC	2000	30,000	5,331	530.4	Yes

[a] This is the average number of changes across the latest 25 revisions.

control software. For example, there are part systems for drive and brake control, interior lightning, exterior lightning, sanitary facilities, HVAC, etc.

The INDUSTRY dataset, with its domain-specific and project-specific concepts, allows us to assess the effectiveness in navigating the noisy, complex, and often irregular nature of real-world data—a critical aspect often overlooked in existing research.

There are currently 8 different trainset types and a common platform comprising 59 different part systems. The total amount of data stored in the model repository amounts to 300GB and the current age of this product line is about 10 years. Since the model revision are maintained in a model repository, we have access to a long history of modeling activities.

The models themselves as well as the average number of changes between revisions in this dataset is large (see Table 5.1) with about 3,300 changes, on average, across the latest 25 revisions. The large number of changes originates from many attributes changes, such as renamings, and typically long time periods between two revisions. The average size is 24.627 *objects*, where an object is an instance of a class in the meta-model, for example, a SoftwareComponent.

Since there are several variants in this product family we can also compare the variants with each other. Comparing across variants leads to 63,096 changes between two variants and for one part system, on average.

5.1.2 RepairVision Dataset

The REPAIRVISION [243, 244] dataset is a public dataset[1] of real-world open-source models, containing histories of 21 ECORE repositories, such as UML2 or BPMN2. Similar to the INDUSTRY dataset, the serialized change graphs in this dataset can become verbose and noisy and reflect the difficulties of real-world model evolution. Its public availability facilitates reproducibility, comparability, and public accessibil-

1 https://repairvision.github.io/evaluation.

ity, fundamental aspects that ensure our research can be examined and extended by others.

The REPAIRVISION dataset helps us to understand to what extent we can use large language models for software model completion[2] in a real-world setting. ECORE is a widely used meta-modeling language in the Eclipse Modeling Framework ecosystem. Furthermore, ECORE is used to model a wide range of domain-specific languages, therefore also increasing the external validity of our study. In particular, the dataset contains UML2 (a widely used software modeling language), SIRIUS (a presentation framework), BPMN2 (a business process modeling language), or BUCKMINSTER (a component build and deploy process automation framework). Listing 5.1 shows an excerpt of the BPMN2 project in the REPAIRVISION dataset in XMI format.

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi
    ="http://www.w3.org/2001/XMLSchema-instance" xmlns:ecore="http://www.
    eclipse.org/emf/2002/Ecore" xmi:id="_c65ugN6tEei97MD7GK1RmA" name="bpmn2"
    nsURI="http://www.omg.org/spec/BPMN/20100524/MODEL-XMI" nsPrefix="bpmn2">
  <eClassifiers xsi:type="ecore:EClass" xmi:id="_c65ugd6tEei97MD7GK1RmA" name="
      DocumentRoot">
    <eAnnotations xmi:id="_c65ugt6tEei97MD7GK1RmA" source="http:///org/eclipse/
        emf/ecore/util/ExtendedMetaData">
      <details xmi:id="_c65ug96tEei97MD7GK1RmA" key="name" value=""/>
      <details xmi:id="_c65uhN6tEei97MD7GK1RmA" key="kind" value="mixed"/>
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EAttribute" xmi:id="
        _c65uhd6tEei97MD7GK1RmA" name="mixed" unique="false" upperBound="-1">
      <eAnnotations xmi:id="_c65uht6tEei97MD7GK1RmA" source="http:///org/eclipse
          /emf/ecore/util/ExtendedMetaData">
        <details xmi:id="_c65uh96tEei97MD7GK1RmA" key="kind" value="
            elementWildcard"/>
        <details xmi:id="_c65uiN6tEei97MD7GK1RmA" key="name" value=":mixed"/>
      </eAnnotations>
      <eType xsi:type="ecore:EDataType" href="plugins_org.eclipse.emf.
          ecore_model_Ecore.ecore#_c658Q96tEei97MD7GK1RmA"/>
    ...
    <eStructuralFeatures xsi:type="ecore:EAttribute" xmi:id="
        _c655Xt6tEei97MD7GK1RmA" name="implementation" ordered="false">
      <eAnnotations xmi:id="_c655X96tEei97MD7GK1RmA" source="http:///org/eclipse
          /emf/ecore/util/ExtendedMetaData">
        <details xmi:id="_c655YN6tEei97MD7GK1RmA" key="kind" value="attribute"/>
        <details xmi:id="_c655Yd6tEei97MD7GK1RmA" key="name" value="
            implementation"/>
```

2 We only use the REPAIRVISION dataset for the model completion experiments, since we did not have this dataset available yet during our study of edit operation mining.

```
    </eAnnotations>
    <eType xsi:type="ecore:EDataType" href="plugins_org.eclipse.emf.
        ecore_model_Ecore.ecore#_c658aN6tEei97MD7GK1RmA"/>
   </eStructuralFeatures>
  </eClassifiers>
</ecore:EPackage>
```

Code Listing 5.1: Extract from bpmn2.ecore in XMI format.

5.1.3 Synthetic Ecore Dataset

The SYNTHETIC dataset is a set of generated models that adhere to an ECORE meta-model, resembling a simplified component model as often used in modeling system architecture. We made this dataset publicly available.[3] With the INDUSTRY and REPAIRVISION datasets, we aimed at external validity and a real-world setting. At the same time, we only have little control over potentially influential factors of the dataset impairing internal validity of conclusions. To obtain a dataset for which we can control several properties of the model repositories, we simulate the evolution of a software model. This dataset gives us control over several properties of a model repository. We mainly use this dataset to understand how qualities of the approach under study are affected by the model repositories' properties.

The repositories in this dataset contain only changes on a type level, that is, we do no include attributes or changes thereof. The meta-model is shown in Figure 2.2. The model repository simulation approach is depicted in Figure 5.1.

For the generation of the dataset, we start with an instance m_0 of the simple component meta-model with 87 Packages, 85 Components, 85 SwImplementations, 172 Ports, 86 Connectors, and 171 Requirements. Then, edit operations are applied e times to the model at random positions (i.e., we select a possible match for the edit operation at random). This leads to a new *revision* m_1. This procedure is applied iteratively d times to obtain the model history

$$m_0 \rightarrow m_1 \rightarrow \dots m_{d-1} \rightarrow m_d.$$

Each evolution step $m_i \rightarrow m_{i+1}$ yields a difference $\Delta(m_i, m_{i+1})$.

The edit operations are randomly chosen from a set of pre-defined edit operations that make sense from a domain point of view. There are three variants of this dataset where either one, two, or three edit operations (depending on the concrete experiment) will be applied to simulate the history.

Furthermore, we also want to control for the level of *noise*. Therefore, to each application of the edit operation, we apply a random perturbation. More concretely, a perturbation is another edit operation that we apply with a certain probability p.

3 https://github.com/se-sic/icse_model_completion

This perturbation is applied such that it overlaps with the application of the main edit operation.

For this dataset, we can therefore control the following parameters for the generated data.

- d: The number of differences in each simulated model repository: $d \in \{10, 20\}$.
- e: The number of edit operations to be applied per model revision in the repository, that is, how often the edit operation will be applied to the model: $e \in \{1, \ldots, 100\}$.
- p: The probability that the operation will be perturbed: $p \in \{0.1, 0.2, \ldots, 1.0\}$.

This gives us 2000 ($= 2 \times 100 \times 10$) simulated repositories. A characteristic of our datasets is that, increasing e, the probability of changes to overlap increases, as well. Eventually, adding more changes even decreases the number of isolated changes while increasing the average size of connected changes.

Technically, to generate the datasets, we use the tool HENSHIN [37] to apply model transformations to a given model revision.

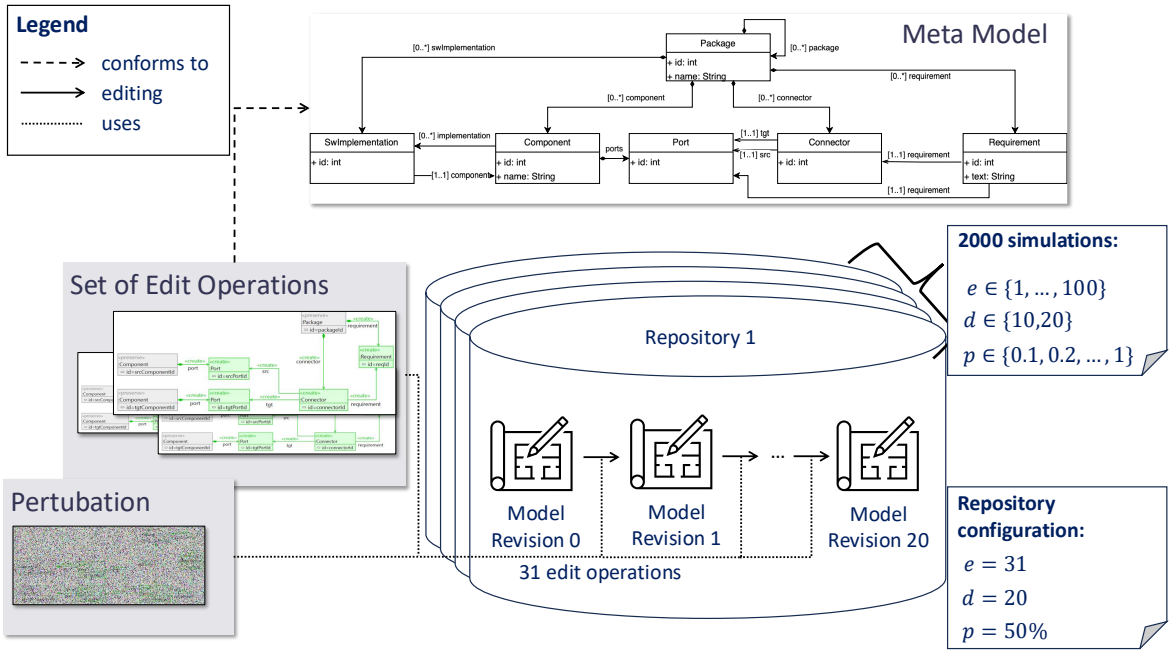

Figure 5.1: Simulation of the model repositories from the Synthetic dataset. A set of predefined edit operations is used to simulate the history of a model repository. A perturbation is applied with a certain probability.

5.2 Processing of the Datasets

Typically, there will be some preprocessing steps that the datasets need to go through. We will describe the common preprocessing and the individual preprocessing of each dataset.

General Preprocessing. As we will see in the following sections, we usually work on the differences between two model revisions. We compute differences between two model revisions. For this end, we transform the model first to models conforming to an ECORE meta-model. We then use the tool SIDIFF [244] to compute the differences between two model revisions. We then extract change graphs from the model differences that we can process in the NETWORKX graph library. The concrete labels and extraction procedure depends on the concrete experiment. We will therefore describe the concrete preprocessing steps in the respective sections.

Also, we typically only use samples from the datasets described above for the concrete experiments. We will describe the sampling procedure in the respective sections.

Furthermore, there is some dataset specific data preprocessing.

Industry Dataset. Since these models describe real-world systems and are subject to confidentiality, we can not make the dataset publicly available. Furthermore, we also can not process the data on a public cloud. For example, for our experiments with large language models, the information in these SYSML models can only be processed in a dedicated Azure deployment of OpenAI models that is certified for the classification level of the data. Furthermore, in a preprocessing step, we have removed confidential information, for example, the models contain requirement owner information and other personal information of involved engineers.

RepairVision Dataset. For some revisions, our preprocessing failed (e.g., because of some inconsistencies in the models). Some revisions and also one entire project had to be excluded from the dataset. This leaves us with the numbers given in Table 5.1, that is, the numbers in this table do not reflect the entire original dataset, but only the results after preprocessing.

Synthetic Dataset. For some experiments, we do consider special variants of the dataset, where we do not apply all three edit operations in the simulation. Depending on the experiment, we apply a different set of edit operations with either one, two, or three edit operations to the model repository. This way, we can control the number of edit operations applied per model revision in the repository.

Part III

Learning from Evolution for Evolution

6

Deriving Edit Operations by Graph Mining

Design isn't finished until somebody is using it.

— Brenda Laurel

This chapter shares material with the Automated Software Engineering Journal 2023 paper titled "Mining Domain-Specific edit operations from Model Repositories with Applications to Semantic Lifting of Model Differences and Change Profiling" [323]. The journal publication is an extension of our earlier work on mining edit operations from model repositories [324] published in the Proceedings of the International Conference on Automated Software Engineering.

Model transformations are central to model-driven software development. Applications of model transformations include creating models, handling model co-evolution, model merging, and understanding model evolution. In particular, edit operations, as a special class of model transformations, constitute the central building blocks of software model evolution. In the past, various (semi-)automatic approaches to derive model transformations from meta-models or from a set of examples have been proposed. These approaches require time-consuming handcrafting or the recording of concrete examples, or they are unable to derive complex transformations. We propose a novel *unsupervised* approach, called OCKHAM, which is able to learn edit operations from model histories in model repositories. OCKHAM employs frequent subgraph mining to discover frequent structures in model difference graphs.

The theory behind OCKHAM has been outlined in Part I of this thesis: meaningful domain-specific edit operations are the ones that *compress* the model differences. In particular Chapter 3 outlines the theoretical background of model evolution and patterns in model evolution. This chapter focuses on the evaluation of these ideas presented in Chapter 3 on parts of the datasets from Chapter 5.

We evaluate our approach in two controlled experiments and one real-world case study of the large-scale industrial model-driven architecture project in the railway domain, presented in Chapter 4. We found that our approach is able to discover frequent edit operations that have actually been applied before. Furthermore, OCKHAM is able to extract edit operations that are meaningful—in the sense of explaining model differences through the edit operations they comprise—to practitioners in an industrial setting. We also discuss use cases (i.e., semantic lifting of model differences and change profiles) for the discovered edit operations in this industrial setting. We find that the edit operations discovered by OCKHAM can be used to better understand and simulate the evolution of models. The results of the evaluation of OCKHAM can also be considered as an empirical evidence for Hypothesis 3 formulated in Chapter 3 that meaningful edit operations can be discovered in model repositories.

In a summary, with this chapter, we make the following contributions:

- We propose an unsupervised approach, called OCKHAM, which is based on frequent subgraph mining to derive edit operations from model repositories, without requiring any further information.
- We evaluate OCKHAM empirically based on two controlled simulated experiments and show that it is able to discover the applied edit operations.
- We evaluate the approach using an interview with five experienced system engineers and architects in the real-world industrial setting described in Chapter 4. We show that our approach is able to detect meaningful edit operations in this industrial setting and that it scales to real-world repositories.
- We apply the automatically derived edit operations to summarize large model differences and show their practical impact in this setting. We furthermore argue how the edit operations can be used to analyze and simulate model evolution.

6.1 Edit Operation Mining: Applications and State-of-the-Art

In this section, we introduce and motivate the problem of edit operation mining that we address in this chapter.

6.1.1 Edit Operations for Model Evolution

In Section 2.4, we defined an *edit operation* as an in-place model transformation. Edit operations usually represent regular evolution [328] of models (cf. Section 2). For example, in UML, when moving a method from one class to another in a class diagram, also a sequence diagram that uses the method in message calls between object lifelines needs to be adjusted accordingly. To perform this in a single edit step, one can perform an edit operation that executes the entire change, including all class and sequence diagram changes.

In Section 3.5, we discussed that edit operations are elementary building blocks in a "generative model" of evolution of software models. In particular, we theoretically derived that selection of edit operations is represented in the model differences by compression of the model differences.

The rest of this chapter will take a more empirical and practical viewpoint on the problem of edit operation mining.

6.1.2 Applications of Edit Operations

Some tasks in Model-driven Engineering can be completely automated and reduced to the definition of edit operations: For example, edit operations are used for the following tasks:

Model repair, quick-fix generation, auto-completion [127, 180, 244]: Edit operations can be used to fix errors, suggest improvements, and complete missing parts of models based on predefined rules or learned patterns.

Model editors [91, 317]: Edit operations can be used to implement model editing commands that can be applied to different types of models, such as class diagrams, sequence diagrams, state machines, etc.

Operation-based merging [178, 285]: Edit operations can be used to merge concurrent changes to models by applying the edit operations that represent the changes in a consistent order.

Model refactoring [20, 234]: Edit operations can be used to improve the quality, structure, and design of models by applying well-known refactoring patterns, such as extracting, renaming, or moving model elements.

Model optimization [46]: Edit operations can be used to optimize the performance, resource consumption, or other criteria of models by applying transformations that reduce the complexity, redundancy, or inefficiency of models.

Meta-model evolution and model co-evolution [19, 111, 132, 184, 270]: Edit operations can be used to handle the changes that occur when a meta-model evolves

and affects the models that conform to it, or when different models need to be synchronized or migrated to a new version of a meta-model.[1]

Semantic lifting of model differences [95, 161, 163, 172, 197]: Edit operations can be used to explain the differences between two model versions in terms of high-level domain concepts and intentions, rather than low-level syntactic changes.

Model generation [258]: Edit operations can be used to generate models from scratch or from existing models by applying transformations that create, modify, or delete model elements according to some criteria or constraints.

6.1.3 Challenges in the Specification of Edit Operations

In general, there are two main problems involved in the specification of edit operations or model transformations in general. Firstly, creating the necessary transformations for the task and the domain-specific modeling languages at hand using a dedicated transformation language requires a deep knowledge of the language's meta-model and the underlying paradigm of the transformation language. It might even be necessary to define project-specific edit operations, which causes a large overhead for many projects and tool providers [155, 160, 234]. Secondly, for some tasks, domain-specific transformations are a form of tacit knowledge [260], and it will be hard for domain experts to externalize this knowledge.

As, on the one hand, model transformations play such a central role in Model-driven Engineering, but, on the other hand, it is not easy to specify them, attempts have been made to support their manual creation or even (semi-)automated generation. As for manual support, visual assistance tools [23] and transformation languages derived from a modeling language's concrete syntax [3, 134] have been proposed to release domain experts from the need of stepping into the details of meta-models and model transformation languages. However, they still need to deal with the syntax and semantics of certain change annotations, and edit operations must be specified in a manual fashion. To this end, generating edit operations automatically from a given meta-model has been proposed [165, 166, 218]. However, besides elementary consistency constraints and basic well-formedness rules, meta-models do not convey any domain-specific information on *how* models are edited. Thus, the generation of edit operations from a meta-model is limited to rather primitive operations as a matter of fact. Following the idea of model transformation by-example [41, 155, 315], initial sketches of more complex and domain-specific edit operations can be specified using standard model editors. However, these sketches require manual post-processing to be turned into general specifications, mainly because an initial

1 In this case, the model transformation is either exogenous (i.e., between different models) or applied on a combined model, including the meta-model of both, source and target model.

specification is derived from only a single transformation example. Some model transformation by-example approaches [160, 234] aim at getting rid of this limitation by using a set of transformation examples as input, which are then generalized into a model transformation rule. Still, this is a *supervised* approach, which requires sets of dedicated transformation examples that need to be defined by domain experts in a manual fashion. As discussed by [160], a particular challenge is that domain experts need to have, at least, some basic knowledge on the internal processing of the model transformation by-example tool to come up with a reasonable set of examples. Moreover, if only a few examples are used as input for learning, [234] discuss how critical it is to carefully select and design these examples.

6.1.4 Motivation: Supporting Large Model Differences

Our initial motivation to automatically mine edit operations from model repositories arose from a collaboration with practitioners from a large-scale industrial model-driven software product line in the railway domain (cf. Chapter 4). Large model differences that need to be handled during several product line engineering activities in this setting have actually been our main motivation to investigate how we can derive edit operations (semi-)automatically:

Discussing major challenges with the engineers of the product line, we observed that some fine-grained model changes appear very often in combination in this repository. For example, when the architect creates an interface between two components, they will usually add some Ports to Components and connect them via the ConnectorEnds of a Connector. Expressed in terms of the meta-model, there are 17 changes to add such an interface (see Figure 4.6). We are therefore interested to automatically detect these patterns in the model repository. More generally, our approach, OCKHAM, is based on the assumption that it should be possible to derive "meaningful" patterns from the repositories. These patterns could then be used for many applications [20, 95, 111, 161, 172, 180, 197, 244, 317].

The background is that, as discussed in Chapter 4, the models have become huge over time (approx. 1.5 million elements split into 59 submodels) and model differences between different products have become huge (63,096 changes in a single submodel, on average). The analysis of these differences, for example, for quality assurance of the models or domain analysis, has become very tedious and time-consuming. To speed-up the analysis of the model differences, it would be desirable to reduce the "perceived" size of the model difference by grouping fine-grained differences to higher-level, more coarse-grained and more meaningful changes. For this *semantic lifting* of model differences, the approach by [163], which uses a set of edit operations as configuration input, can be used but the approach requires the edit operations to be defined already. Based on a set of edit operations this semantic lifting approach will group changes into so called *change sets*.

We will use the data from this real-world project to evaluate OCKHAM in Section 6.3.

6.1.5 Unsupervised Mining of Edit Operations

To address these limitations of existing approaches, we propose a novel *unsupervised* approach, OCKHAM, for mining edit operations from existing models in a model repository, which is typically available in large-scale modeling projects (cf. Chapter 4). OCKHAM is based on an Occam's razor argument (cf. Principle 3.2.1), that is, the *useful* edit operations are the ones that *compress* the model repository. In a first step, OCKHAM discovers frequent change patterns using *frequent subgraph mining* on a labeled graph representation of model differences. It then uses a *compression metric* to filter and rank these patterns.

We evaluate OCKHAM on a selection of the datasets from Chapter 5, that is, two controlled experiments with data from the SYNTHETIC and one real-world large-scale industrial case study with data from the INDUSTRY dataset. In the controlled setting, we can show that OCKHAM is able to discover the edit operations that have been actually applied before by us in the simulation of the data, even when we apply some perturbation as a kind of *noise*. In the real-world case study, we find that our approach is able to scale to real-world model repositories and to derive edit operations deemed reasonable by practitioners. We evaluated OCKHAM by comparing the results to randomly generated edit operations in five interviews with practitioners of the product line. We find that the edit operations represent typical edit scenarios and are meaningful to the practitioners. Additionally, we evaluate the practical applicability of the derived edit operations based on a concrete real-world scenario. More specifically, motivated by our aim to facilitate the analysis of model differences, we evaluate the extent to which the edit operations can be used to compress and filter model differences. Second, we discuss key observations that we made in the case study when further analyzing model differences after a semantic lifting using the edit operations mined by OCKHAM. In particular, we have found that the frequency distribution of the edit operations in the model differences gives rise to a change profile that describes the model difference from a statistical perspective and can be used for classifying the model differences. We discuss applications of these profiles, including the statistical analysis of model differences and the use of the change profiles to simulate model evolution.

6.2 Approach

In this section, we present an approach to solve the problem of inferring edit operations from a model repository (see Definition 3.5.1) based on Hypothesis 3 and the minimum description length principle 3.2.1.

It is worthwhile mentioning that the theory provided in Chapter 3 is not constructive. That is, the minimum description length principle 3.2.1 is rather declarative and does not provide a constructive way to find the edit operations that lead to the minimal description length. The theory requires to provide a family of models[2] $\mathcal{H}$, which in our case would be the sets of edit operations. It then provides a selection criterion, to select "good" candidate sets of edit operations from $\mathcal{H}$. Of course, for most meta-models, the family of all possible sets of edit operations is too large to be computed and evaluated in a brute-force manner. Therefore, we need to come up with a constructive approach to find the edit operations that (approximate) a minimal description length.

We address the problem of automatically identifying edit operations, formally defined in Definition 3.5.1, from a graph mining perspective. As discussed in Section 2, we will work with labeled graphs instead of typed graphs. There are some limitations related to this decision, which we discuss in Section 6.4.3.

OCKHAM consists of the five steps illustrated with a running example in Figure 6.1–6.4. Our main *technical* contributions are Step 2 and Step 4. For Step 1, Step 3, and Step 5 we apply existing tooling: SIDIFF, GASTON, and HENSHIN (cf. Section 2 and Section 3.3).

Step 1: Matching and Derivation of Difference Graphs: To learn a set of edit

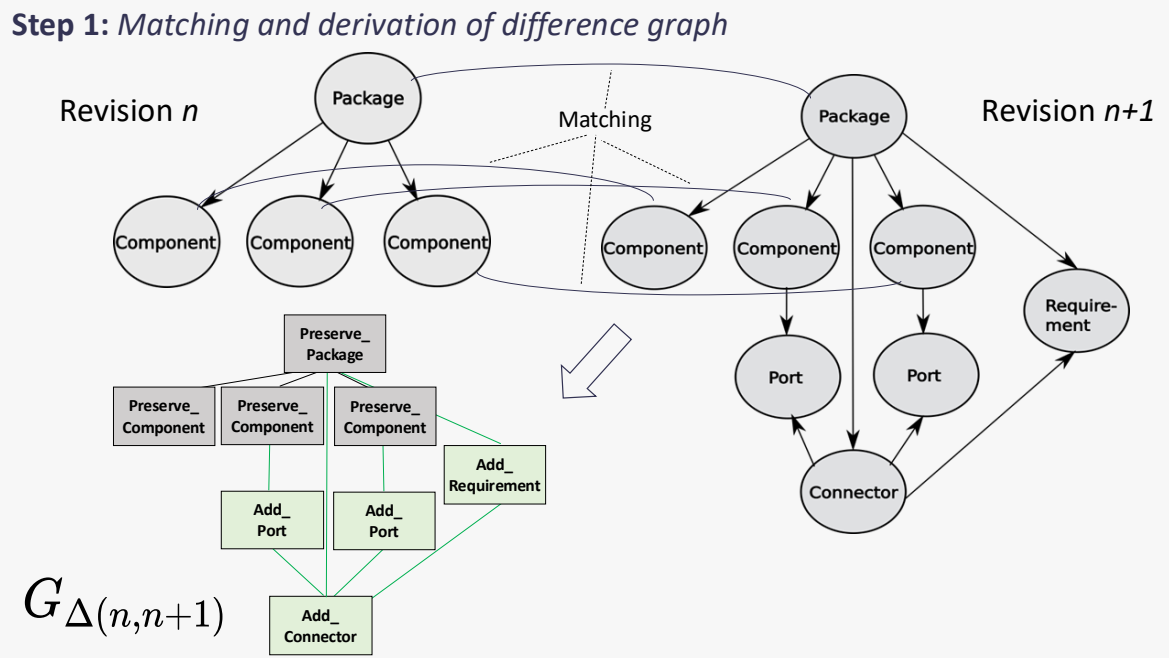

Figure 6.1: The 5-step process for mining edit operations with OCKHAM. First, in Step 1, a structural model difference is computed between two model versions.

operations in an *unsupervised* manner, OCKHAM analyzes model changes that can be

2 In the sense of a model to explain the data and not to be confused with software models.

extracted from a model's development history. For every pair of successive model versions n and $n+1$ in a given model history, we calculate a *structural model difference* $\Delta(n, n+1)$ (cf. Definition 2.4.3) to capture these changes. As we do not assume any information (e.g., persistent change logs) to be maintained by a model repository, we use a state-based approach to calculate a structural difference, which proceeds in two steps [159]. First, the corresponding model elements in the model graphs G_n and G_{n+1} are determined using a model matcher [182]. In many cases, the model elements will carry identifiers, which we can utilize in the matching. Second, the structural changes are derived from these correspondences: All the elements in G_n that do not have a corresponding partner in G_{n+1} are considered to be deleted, whereas, vice versa, all the elements in G_{n+1} that do not have a corresponding partner in G_n are considered to be newly created.

For further processing in subsequent steps, we represent a structural difference $\Delta(n, n+1)$ in a graph-based manner, referred to as *difference graph* [244]. A difference graph $G_{\Delta(n,n+1)}$ is constructed as a unified graph over G_n and G_{n+1}. That is, corresponding elements being preserved by an evolution step from version n to $n+1$ appear only once in $G_{\Delta(n,n+1)}$ (indicated by the label prefix[3] "Preserve_"), while all other elements that are unique to model G_n and G_{n+1} are marked as deleted and created, respectively (indicated by the label prefixes "Add_" and "Remove_").

For illustration, assume that the architectural model shown in Figure 2.2 is the revised version $n+1$ of a version n by adding the ports along with the connector and its associated requirement. Figure 6.1 illustrates a matching of the abstract syntax graphs of the model versions n and $n+1$. For the sake of brevity, only correspondences between nodes in G_n and G_{n+1} are shown in the figure, while two edges are corresponding when their source and target nodes are in a correspondence relationship. The derived difference graph $G_{\Delta(n,n+1)}$ is illustrated in Figure 6.1. For example, the corresponding nodes of type Component occur only once in $G_{\Delta(n,n+1)}$, and the nodes of type Port are indicated as being created in version $n+1$.

Our implementation is based on the Eclipse Modeling Framework. We use the tool SiDiff [162, 284] to compute structural model differences. Our requirements on the model differencing tool are: (1) support for EMF, (2) the option to implement a custom matcher, because modeling tools such as MagicDraw usually provide IDs for every model element, which can be employed by a custom matcher, and (3) an approach to semantically lift model differences based on a set of given edit operations, because we intend to use the semantic lifting approach for the compression of differences in the industrial case study from Chapter 4. Other tools such as EMFCompare could also be used for the computation of model differences and there are no other criteria to favor one over the other. An overview of the different

3 Instead of label prefixes, one could also indicate the change information as additional attribute or extra meta data. For later experiments, instead of having "flat" label representations, we used JSON label representations.

matching techniques is given by [182]; a survey of model comparison approaches is given by [313].

Step 2: Derivation of Simple Change Graphs (SCG): Real-world models maintained

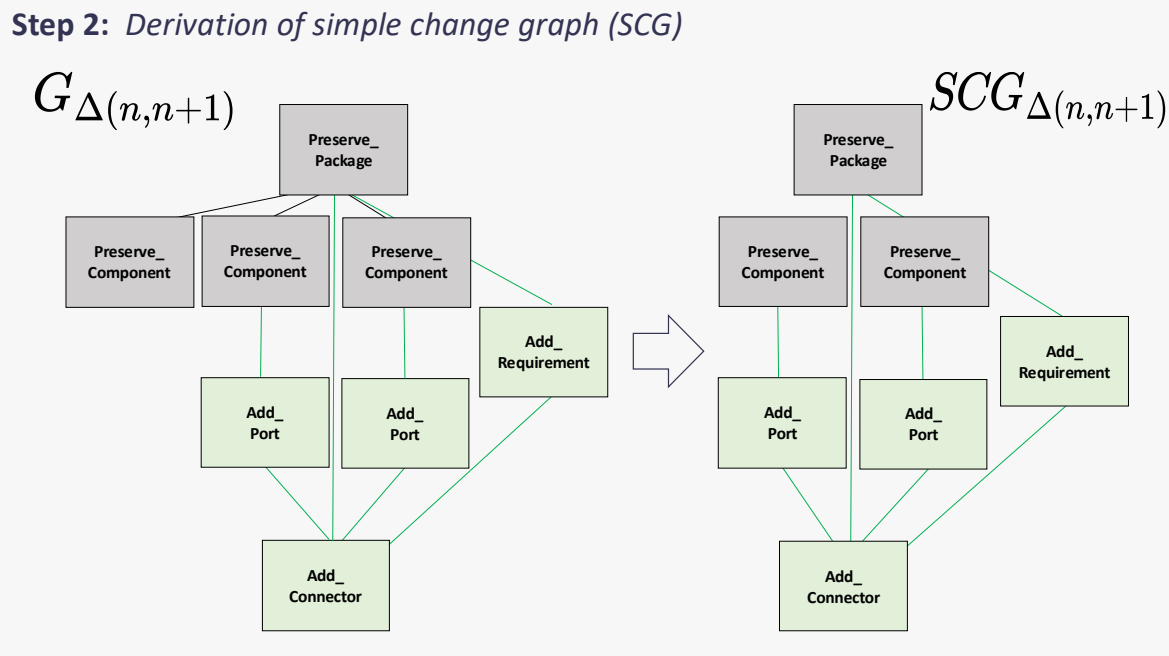

Figure 6.2: The 5-step process for mining edit operations with OCKHAM. In Step 2, a simple change graph is derived from the structural model difference.

in a model repository, such as the architectural models in our case study, can get huge. It is certainly fair to say that, compared to a model's overall size, only a small number of model elements is actually subject to change in a typical evolution step. Thus, in the difference graphs obtained in the first step, the majority of difference graph elements represent model elements that are simply preserved. To this end, before we continue with the frequent subgraph mining in Step 3, in Step 2, difference graphs are reduced to *simple change graphs* (SCGs) based on the principle of *locality relaxation*: only changes that are "close" to each other can result from the application of a single edit operation. We discuss the implications of this principle in Section 6.4.3. By "close", we mean that the respective difference graph elements representing a change must be directly connected (i.e., via only one edge in the graph). Conversely, this means that changes being represented by elements that are part of different connected components of a simple change graph are independent of each other (i.e., they are assumed to result from different edit operation applications).

The definition of simple change graphs is given in Definition 2.4.4. Here, we will present a constructive derivation: given a difference graph $G_{\Delta(n,n+1)}$, we derive a simple change graph $SCG_{\Delta(n,n+1)}$ (cf. Definition 2.4.4) from $G_{\Delta(n,n+1)}$ in two steps. First, we select all the elements in $G_{\Delta(n,n+1)}$ representing a change (i.e., nodes and

edges that are labeled as "Remove_*" and "Add_*", respectively). In general, this selection does not yield a graph, but just a graph fragment $F \subseteq G_{\Delta(n,n+1)}$, which may contain dangling edges. Second, preserved nodes adjacent to dangling edges are also selected to be included in the simple change graph. Formally, the simple change graph is constructed as the boundary graph of F, which is the smallest graph $SCG_{\Delta(n,n+1)} \subseteq G_{\Delta(n,n+1)}$ completing F to a graph [159]. The derivation of a simple change graph from a given difference graph is illustrated in Figure 6.2. In this example, the simple change graph comprises only a single connected component. In a realistic setting, however, a simple change graph typically comprises a larger set of connected components, like the one illustrated in Step 3 in Figure 6.3. The simple change graph can also be considered as a blueprint for the edit operation. It contains all the changes performed by the edit operation and the "anchors" it needs in the current model.

We implemented a generator for the simple change graphs in EMF. In our implementation, we loop through all the changes in the symmetric difference and for each change c, we add nodes and edges as below[4]:

`AddNode`: A node with label `Add_{TypeName}` is added to the graph for an `AddNode` change.

`RemoveNode`: A node with label `Remove_{TypeName}` is added to the graph for a `RemoveNode` change.

`AddEdge`: For a change of type `AddEdge`, the source object and the target object of the added reference have to be either preserved or also been added to the model m_{i+1}. If the source/target node have not yet been added to the graph, we add a node v_{source}/v_{target} with the labels `Add_{TypeName}` or `Preserve_{TypeName}`, respectively. We then add an edge $e = (v_{source}, v_{target})$ between source and target node to the edge set E of the graph.

`RemoveEdge`: We add an edge for a change c of type `RemoveEdge` but the source and target nodes can be either removed or preserved in this case.

`AttributeValueChange`: An attribute value needs to belong to a preserved object in the model. The corresponding `Preserve_{TypeName}` node is added to the graph if it is not yet there. A node with label `Change_{AttributeName}_On_{TypeName}` is then added to the graph. The preserved node and the node representing the changed attribute will then be connected by an edge with the same label.

Details of the derivation of the simple change graphs are given in Section E.3.1 in the appendix.

Step 3: Apply Frequent Subgraph Mining: Using the previous two steps for a given

4 In the node and edge label, we omit attribute information. The edit operations is therefore rather a type blueprint, and we do not extract attribute patterns in this approach. Depending on the serialization we choose, we can determine the degree of freedom in the attribute representation. Since we use "exact" frequent subgraph mining, in this approach, we will only work on a type level. For a discussion of the label representation, see Section 2 and Section E.3.1. We will discuss limitations of the approach presented here in Section 6.4.3.

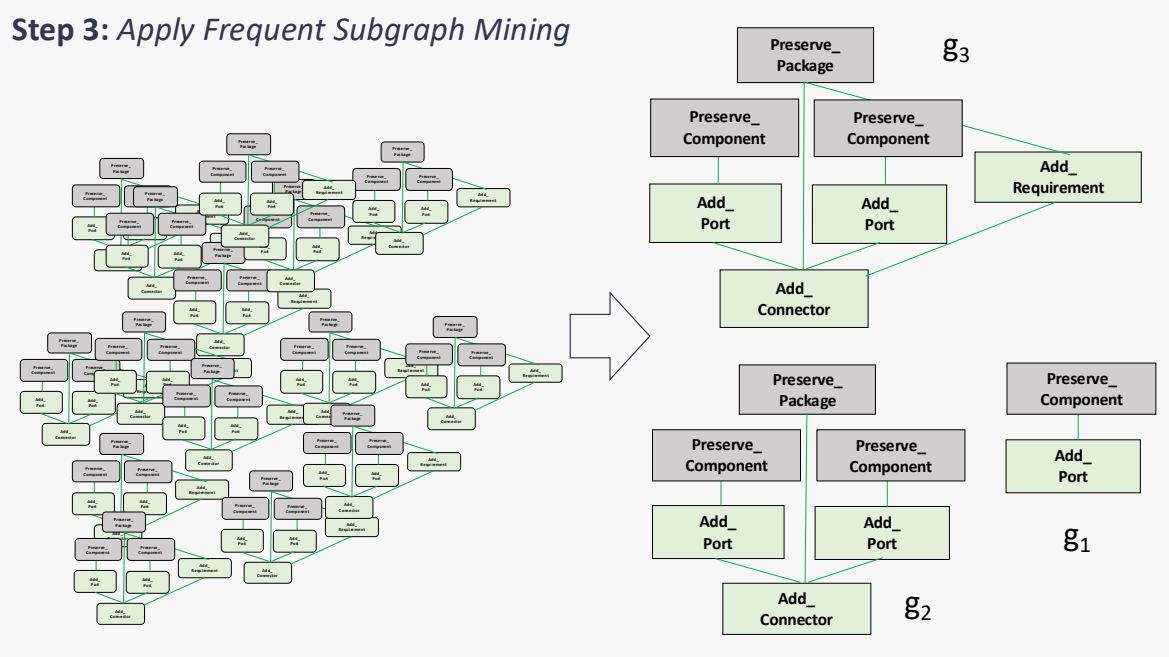

Figure 6.3: The 5-step process for mining edit operations with OCKHAM. In Step 3, frequent connected subgraphs are mined from the simple change graphs.

model repository that contains a set of model revisions, we can compute a set of Simple Change Graphs. These change graphs contain information about fine-grained, meta-model-based changes between two successive revisions of models. We believe that these fine-grained changes can be grouped to more coarse-grained changes. In the example of Figure 6.1, a connector, together with its ports and a requirement, are added. The underlying idea of the approach is that changes that can be *grouped together* should also frequently *appear together* in our model differences. When we apply the first two steps to a model history, we obtain a set of simple change graphs

$$\{SCG_{\Delta(n,n+1)} \mid n \in \{1,\ldots,N-1\}\},$$

where N is the number of revisions in the repository. In this set, we want to identify recurring patterns and therefore find some frequent connected subgraphs. A small support threshold might lead to a huge number of frequent subgraphs. For example a support threshold of one would yield every subgraph in the set of connected components. This does not only cause large computational effort, but also makes it difficult to find relevant subgraphs.

We therefore use some heuristics to determine a suitable support threshold described in detail in Section D.1.3 in the appendix. Basically this heuristic is a function

$$t : D \to \mathbb{N},$$

that maps a given (graph) dataset D to a support threshold for the (exact) frequent subgraph mining. We discuss the effect of the support threshold further in Section 6.4.

We run the frequent subgraph miner for the threshold $t(D)$ found via the heuristic. Step 3 in Figure 6.3 shows this for our running example. We start with a set of connected components, and the graph miner returns a set of frequent subgraphs, namely $\{g_1, g_2, g_3\}$ with $g_1 \subset g_2 \subset g_3$.

We use the GASTON [242] graph miner, since it performed best (in terms of runtime) among the miners that we experimented with (GSPAN, GASTON, and DIMSPAN) in a pilot study. In that pilot study, we ran the miners on a small selection of our datasets and experimented with the parameters of the miners. For many datasets, GSPAN and DIMSPAN did not terminate at all (we canceled the execution after 48h). GASTON (with embedding lists) was able to terminate in less than 10s on most of our datasets but consumes a lot of memory, typically between 10 GB–25 GB, which was not a problem for our 32 GB machine in the pilot study. To rule out any effects due to approximate mining, we considered only exact miners. Therefore, we also could not use SUBDUE [170], which directly tries to optimize compression. Furthermore, SUBDUE was not able to discover both edit operations in the second experiment (see Section 6.3), without iterative mining and allowing for overlaps. Enabling these two options, SUBDUE did not terminate on more than 75% of the pilot study datasets

Step 4: Select Most Relevant Subgraphs (Ranking): Motivated by the *minimum*

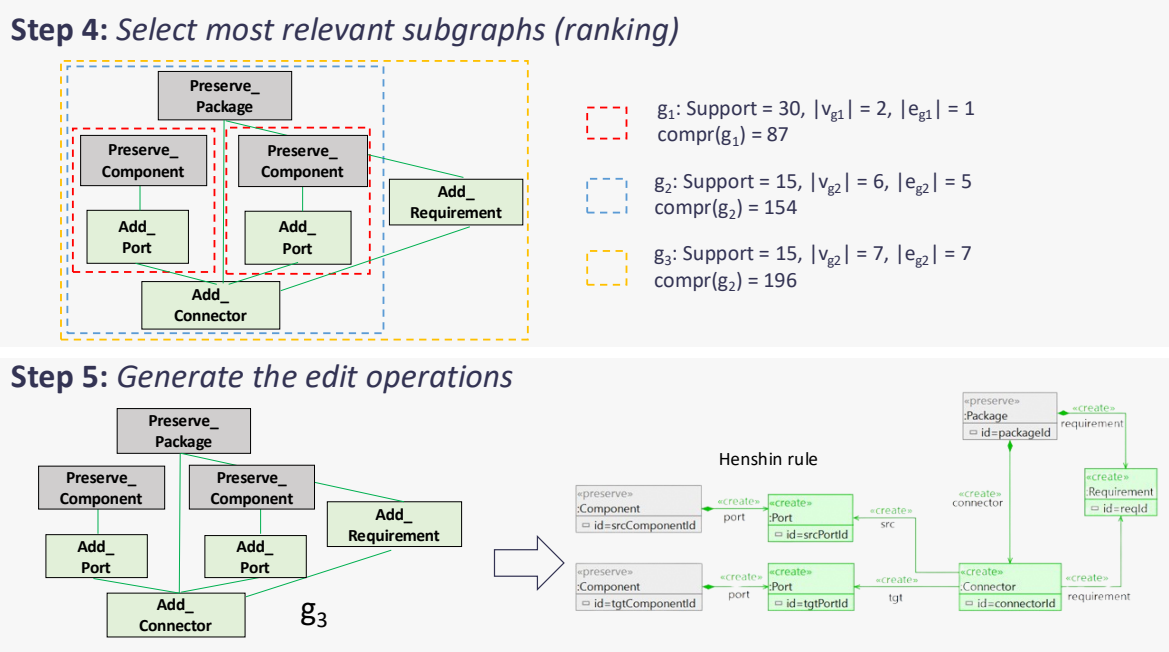

Figure 6.4: In Step 4, the mined subgraphs are filtered and ranked based on a compression criterion. Finally, in Step 5, the edit operations are generated from the mined subgraphs.

description length principle (cf. Principle 3.2.1), which has been successfully applied to many kinds of data [118], the most relevant patterns should not be the most frequent

ones but the ones that give us a maximum compression for our original data [83]. That is, we want to express the given SCGs by a set of subgraphs such that the description length for the subgraphs together with the length of the description of the SCGs in terms of the subgraphs becomes minimal. A detailed motivation for this approach is given in Section 3.5, and we here only want to give a brief overview of the argument by looking at the corner cases: (1) A single change has a large frequency but is typically not interesting. (2) The entire model difference is large in terms of changes but has a frequency of only one and is typically also not an interesting edit operation. "Typical edit operations" are therefore somewhere in the middle. We will use our experiments in Section 6.3 to validate whether this assumption holds.

Definition 6.2.1 Compression Value.
We define the compression *value by*

$$\text{compr}(g) = \big(\text{supp}(g) - 1\big) \cdot \big(\mid V_g \mid + \mid E_g \mid \big)\,, \tag{6.1}$$

where $\text{supp}(g)$ *is the support of* g *in our set of input graphs (i.e., the number of components in which the subgraph is contained).*

This definition is equivalent to Equation 3.23 for subgraphs, with the size replaced by the number of nodes plus the number of edges and without the scaling factor (which is only necessary to compare across different datasets). The "-1" in the definition of the compression value comes from the intuition that we need to store the definition of the subgraph, to decompress the data again. The goal of this step is to detect the subgraphs from the previous step with a high compression value. Subgraphs are organized in a *subgraph lattice*, where each graph has pointers to its direct subgraphs. Most of the subgraph miners already compute a subgraph lattice, so we do not need a subgraph isomorphism test here. Due to the downward closure property of the support, all subgraphs of a given (sub-)graph have, at least, the same frequency (in transaction-based graph mining). When sorting the output, we need to take this into account, since we are only interested in the largest possible subgraphs for some frequency. These largest possible subgraphs for some frequency are called closed subgraphs and typically the number of closed subgraphs is much smaller than the number of all frequent subgraphs [356]. Therefore, we prune the subgraph lattice. The resulting list of recommendations is then sorted according to the compression value. Other outputs are conceivable, but in terms of evaluation, a sorted list is a typical choice for a recommender system [288].

More technically, let SG be the set of subgraphs obtained from Step 3, we then remove all the graphs in the set

$$\begin{aligned} SG^{-} = \big\{ g \in SG \mid \exists\, \tilde{g} \in SG, \text{with } g \subseteq \tilde{g} \\ \wedge\, \text{supp}(g) = \text{supp}(\tilde{g}) \wedge \text{compr}(g) \leq \text{compr}(\tilde{g}) \big\}. \end{aligned}$$

Our list of recommendations is then $SG \setminus SG^{-}$, sorted according to the compression metric.

For our running example in Step 4 in Figure 6.4, assume that the largest subgraph g_3 occurs 15 times (without overlaps). Even though the smaller subgraph g_1 occurs twice as often, we find that g_3 provides the best compression value and is therefore ranked first. Subgraph g_2 will be pruned, since it has the same support as its supergraph g_3, but a lower compression value.

We implement the compression computation and pruning using the *NetworkX*[5] Python library. The algorithm outputs a compression-ranked and pruned list of frequent subgraphs.

Step 5: Generate Edit Operations: As a result of Step 4, we have an ordered list of "relevant" subgraphs of the simple change graphs. We need to transform these subgraphs into model transformations that specify the mined edit operations. As illustrated in Step 5 in Figure 6.4, the subgraphs can be transformed to HENSHIN transformation rules in a straightforward manner. The added elements are added to the right-hand side graph of the HENSHIN transformation rule, the deleted elements are added to the left-hand side graph of the HENSHIN transformation rule, and preserved elements are added to both graphs and a corresponding matching is added to the rule. At this point, we can also use the information from the meta-model to complete the edit operations. For example, when a created element requires another element or attribute, which is not yet present, it can be added. This is done similarly to the derivation of consistency preserving edit operations by Kehrer et al. [166]. A domain expert can add some (non-)application conditions to the rules or add further parameters for the rules. We use HENSHIN because it is used for the semantic lifting approach in our case study from Chapter 4. In principle, any transformation language that allows us to express endogenous, in-place model transformations could be used. A survey of model transformation tools is given by Kahani et al. [154].

6.3 Evaluation

In Chapter 3, we have developed a theory that formally connects patterns in a software model repository to the description—leveraging the patterns—of the very same model repository. We formulated as Hypothesis 3 that those patterns that can be used to compress the model repositories correspond to meaningful edit operations. One of the ideas behind this hypothesis is that having the corresponding edit operations at hand would have led to a minimal effort path to derive at the current state of models. We refer the reader to Chapter 3 for an in-depth discussion of the theory behind this hypothesis. We intentionally formulated Hypothesis 3 as a *hypothesis*, since it is not clear that this optimization criterion will also lead to

5 https://networkx.org/

meaningful edit operations in practice. That is, even though a set of edit operations might have been the optimal path to derive the current state of the models, they might rather constitute a shortcut that is hard to understand for a human. Additional to the theoretical considerations and the correspondence to Assembly Theory in Theorem 3.5.3, we therefore have to come up with an empirical justification of the hypothesis.

In this section, we will evaluate the approach from Section 6.2 in two controlled experiments (with data from the SYNTHETIC dataset) and one real-world industry case study in the railway domain (with data from the INDUSTRY dataset).

6.3.1 Research Questions for Experimental Context

We evaluate OCKHAM w.r.t. the following research questions:

RQ 1: *Is* OCKHAM *able to identify edit operations that have actually been applied in model repositories?*

If we apply some operations to models, OCKHAM should be able to discover these from the data. Furthermore, when different edit operations are applied and overlap, it should still be possible to discover them.

RQ 2: *Is* OCKHAM *able to find typical edit operations or editing scenarios in a real-world setting?*

Compared to the first research question, OCKHAM should also be able to find typical scenarios in practice, in scenarios where we do not know which operations have been actually applied to the data. Furthermore, it should be possible to derive these edit operations in a real-world setting with large models and complex meta-models.

RQ 3: *What are the main drivers for* OCKHAM *to succeed or fail?*

We want to identify the characteristics of the input data and parameters having a major influence on OCKHAM.

RQ 4: *What are the main parameters for the performance of the frequent subgraph mining?*

Frequent subgraph mining has a very high computational complexity for general cyclic graphs. We want to identify the characteristics of the data in our setting that influence the mining time.

RQ 5: *What is the practical impact of the edit operations discovered by* OCKHAM *in the context of the industrial case study from Chapter 4?*

One of our main motivations is to compress differences in our case study from Chapter 4. We evaluate what degree of compression our mined operations provide in this scenario.

For RQ 1, we want to rediscover the edit operations from our ground truth, whereas in RQ 2, the discovered operations could also be some changes that are not applied in "only one step" but appear to be typical for a domain expert. It is certainly fair to refer to both of these cases as "meaningful" edit operations, therefore these research questions are promising to investigate Hypothesis 3.

6.3.2 Experiment Setup

We conduct four experiments to evaluate our approach. In the first two experiments, we run the algorithm on our SYNTHETIC dataset. We know the "meaningful" edit operations in these repositories, since we defined them, and apply them to sample models. We can therefore use these experiments to answer RQ 1. Furthermore, since we are able to control many properties of our input data for these simulated repositories, we can also use them to answer RQ 3 and RQ 4. In the third experiment, we apply OCKHAM to the INDUSTRY dataset to answer RQ 2. The first two experiments help us find the model properties and the parameters the approach is sensible to. Their purpose is to increase the *internal validity* of our evaluation. To increase *external validity*, we apply OCKHAM in a real-world setting as well. None of these experiments alone achieves sufficient internal and external validity [302], but the combination of all experiments is suitable to assess whether OCKHAM can discover relevant edit operations.[6]

In the fourth experiment, we use the edit operations from the third experiment to semantically lift the low-level differences using an approach for semantic lifting of model difference by Kehrer et al. [163]. We then compute the compression ratio on a sample of model differences to answer RQ 5.

We run the experiments on an Intel Core™ i7-5820K CPU @ 3.30GHz × 12 and 31.3 GiB RAM. For the synthetic repositories, we use 3 cores per dataset.

Experiment 1: As a first experiment, we simulate the application of edit operations on a simple component model. The meta-model is shown in Figure 2.2. We apply OCKHAM to the "simplest" variant of the SYNTHETIC dataset: We only apply one kind of edit operation (the one from our running example in Figure 6.1–6.4) to a random model instance. The Henshin rule specifying the operation consists of a

6 At the time of conducting the experiments, we did not have access to the REPAIRVISION dataset. We therefore do not include this dataset in the evaluation of OCKHAM.

graph pattern comprising 7 nodes and 7 edges. We create the model differences as follows: We start with an instance m_0 of the simple component meta-model with 87 Packages, 85 Components, 85 SwImplementations, 172 Ports, 86 Connectors, and 171 Requirements. Then, the edit operation is randomly applied e times to the model obtaining a new model revision m_1. This procedure is applied iteratively d times to obtain the model history $m_0 \rightarrow m_1 \rightarrow \ldots m_{d-1} \rightarrow m_d$. Each evolution step $m_i \rightarrow m_{i+1}$ yields a difference $\Delta(m_i, m_{i+1})$.

Since we can not ensure completeness of OCKHAM (i.e., it might not discover all edit operations in a real-world setting), we also have to investigate how sensible the approach is to undiscovered edit operations. Therefore, to each application of the edit operation, we apply a random perturbation. More concretely, a perturbation is another edit operation that we apply with a certain probability p. This perturbation is applied such that it overlaps with the application of the main edit operation. We use the tool HENSHIN [37] to apply model transformations to one model revision. We then build the difference of two successive models as outlined in Section 6.2. In our experiment, we control the parameters as described in Chapter 5.

OCKHAM suggests a ranking of the top k subgraphs (which eventually yield the learned edit operations). In the ranked suggestions of the algorithm, we then look for the position of the "correct" edit operation (i.e., the one we applied in the simulation of the history) by using a graph isomorphism test.

Definition 6.3.1 MAP@k.
The mean average precision at k *(MAP@k) is defined as the mean of the average precision at k (AP@k) over all datasets. To evaluate the ranking, we use the "mean average precision at k" (MAP@k), which is commonly used as an accuracy metric for recommender systems [288]:*

$$\text{MAP@k} := \frac{1}{|D|} \sum_{D} \text{AP@k} \ ,$$

where D is the family of all datasets (one dataset represents one repository) and AP@k is defined by

$$\text{AP@k} := \frac{\sum_{i=1}^{k} \text{P}(i) \cdot \text{rel}(i)}{|\textit{total set of relevant subgraphs}|} \ ,$$

where P(i) is the precision at i, and rel(i) indicates if the graph at rank i is relevant.

For this experiment, the number of relevant edit operations (or subgraphs to be more precise) is always one. Therefore, we are interested in the rank of the correct edit operation. Except for the case that the relevant edit operation does not show up at all, MAP@∞ gives us the mean reciprocal rank and therefore serves as a good metric for that purpose.

For comparison only, we also compute the MAP@k scores for the rank of the correct edit operations w.r.t. the frequency of the subgraphs. Furthermore, we investigate how the performance of subgraph mining depends on other parameters of OCKHAM. We are also interested in how average precision (AP), that is, AP@∞, depends on the characteristics of the datasets. Note that for the first two experiments, we do not need to execute the last canonical step of our approach (i.e., deriving the HENSHIN transformation rule from a SCG subgraph). Instead, we directly evaluate the resulting subgraph from Step 4 against the simple change graph corresponding to the edit operation.

To evaluate the performance of the frequent subgraph miner on our datasets, we fixed the relative threshold (i.e., the support threshold divided by the number of components in the graph database) to 0.4. We re-run the algorithm for this fixed relative support threshold and $p \leq 0.4$ (i.e., we disable the heuristic to determine the support threshold from the dataset).

Experiment 2: In contrast to the first experiment, in the second experiment, we want to identify more than one edit operation in a model repository. We therefore extend the first experiment by adding another edit operation, applying each of the operations with the same probability (i.e., we use the SYNTHETIC dataset with two edit operations). To test whether OCKHAM also detects edit operations with smaller compression than the dominant (in terms of compression) edit operation, we choose a smaller second operation. The Henshin rule graph pattern for the second operation comprises 4 nodes and 5 edges. It corresponds to adding a new Component with its SwImplementation and a Requirement to a Package. Since the simulation of model revisions consumes a lot of compute resources[7], we fixed $d = 10$ and considered only $e <= 80$ for this experiment. The rest of the experiment is analogous to the first experiment.

Experiment 3: The power of the simulation to mimic a real-world model evolution is limited. Especially, the assumption of random and independent applications of edit operations is questionable. Therefore, for the third experiment, we use a real-world model repository from the railway software development domain (i.e., the INDUSTRY dataset). For this repository, we do not know the operations that have actually been applied. We therefore compare the mined edit operations with edit operations randomly generated from the meta-model, and want to show that the mined edit operations are significantly more "meaningful" than the random ones.

7 This high utilization of compute resources is mainly due to the application of noise. Since we wanted to apply the noise without risking a bias, we just randomly computed matches of the noise rules, until they overlap with the already applied edit operation. For large models this needs a significant amount of reshuffling. Certainly, there is a more efficient way to apply the noise. Anyway, ensuring there is no bias in the overlap would require a more sophisticated approach, and we wanted to avoid overengineering the simulation.

The models in this case are SysML models in MagicDraw, but there is an EMF export of the SysML models, which we can use to apply our toolchain.

For this experiment, from the INDUSTRY dataset, we sampled 546 pairwise differences, with 4109 changes, on average, which also contain changed attribute values (one reason for that many changes is that the engineering language has changed from German to English). The typical model size in terms of their abstract syntax graphs is 12081 nodes; on average, 50 out of 83 meta-model classes are used as node types. In this sampling, we selected only a subset of the part systems and variants, namely those that the domain experts in our interviews were familiar with. From these, we randomly selected 546 revisions and computed the difference to their predecessor.

To evaluate the quality of our recommendations, we conducted a semi-structured interview with five domain experts of our industry partner: 2 system engineers working with one of the models, 1 system engineer working cross-cutting, 1 chief system architect responsible for the product line approach and the head of the tool development team. We presented them 25 of the top-ranked (according to the compression metric from Equation 6.1) mined edit operations together with 25 edit operations that were randomly generated out of the meta-model. Duplicate edit operations and edit operations that result from multi-object structures were filtered out before presenting the results to the domain experts (e.g., if the recommendations included edit operations for adding two activity diagram `swimlanes`, adding three `swimlanes`, etc., we only included one of them).

We only relied on the compression-based ranking since it is arguably superior to the purely frequency-based ranking, as we have seen empirically in the first two experiments. The edit operations were presented in the visual transformation language of HENSHIN, which we introduced to our participants before. On a 5-point Likert scale, we asked whether the edit operation represents a typical edit scenario (5), is rather typical (4), can make sense but is not typical (3), is unlikely to exist (2), and does not make sense at all (1). We compare the distributions of the Likert score for the population of random edit operations and mined edit operations to determine whether the mined operations are typical or meaningful.

In addition, we qualitatively discussed the mined edit operations that have not been considered to be typical with the engineers.

Experiment 4: Higher-level edit operations typically comprise several more fine-grained edit steps. Kehrer et al. [163] presented an approach that allows to represent a model difference in terms of previously defined edit operations. Changes in the model difference that belong to the application of one edit operation are grouped together in a *lifted model difference*. A group of changes that belong to the application of one edit operation is called a *change set*. Using this lifting approach, we can evaluate to what extend the edit operations mined by OCKHAM can compress model differences. For this, in the fourth experiment, we use the 25 mined edit operations from the third experiment and five additional edit operations discovered by OCKHAM for a

semantic lifting of a sample of 40 model differences from the INDUSTRY dataset (cf. Section 5.1.1). More concretely, we randomly select 40 submodels in their current revision and compute the difference between this revision and 25 revisions earlier of the same submodel. If no 25 revisions are available, we compute the difference to the earliest possible revision. Of course, our 30 edit operations are not yet complete, that is, some changes will be missed. We will therefore not only report the total compression ratio but also the relative compression ratio (i.e., the compression ratio with respect to the changes that are covered by the 30 edit operations). We define the compression ratio as follows:

Definition 6.3.2 Compression Ratio.
The compression ratio *is defined as the ratio of the number of changes before and after the semantic lifting.*

$$c := \frac{|\textit{changes before lifting}|}{|\textit{change sets}| + |\textit{ungrouped changes}|},$$

where by ungrouped changes *we refer to changes that are not included in any of the change sets after the semantic lifting. Similarly, the relative compression ratio is defined as:*

$$c_{\text{rel}} := \frac{|\textit{changes before lifting}| - |\textit{ungrouped changes}|}{|\textit{change sets}|}$$

With the help of the domain experts we furthermore divided the set of 30 edit operations into two subsets. One subset contains *refactorings* like renaming operations or move operations, while the other subset contains edit operations which represent important *functional evolution*. We will refer to the second subset as *critical edit operations*. 22 edit operations have been classified as critical and 8 as refactorings. We will also report the compression ratio for these critical operations separately. The reason for this separate reporting is that for some tasks related to model differences, refactoring is less relevant and can be ignored, while the critical edit operations are crucial for the analysis. An example for such a task is the analysis of the functional changes of a product between two successive software qualifications. It is important to emphasize here that the discovery of refactoring edit operations is still important and a prerequisite for this filtering.

6.3.3 Results

Experiment 1: In Table 6.1, we list the MAP@k scores for all datasets in the experiment. Table 6.3 shows the Spearman correlation of the independent (i.e., perturbation p, applied edit operation count e, and number of revisions d) and dependent (i.e., mining time, average precision, and mean number of nodes per connected com-

ponent) variables. If we look only on datasets with a large number of applied edit operations, $e > 80$, the Spearman correlation for average precision vs. d and average precision vs. p becomes 0.25 (instead of 0.12) and -0.14 (instead of -0.07), respectively. The mean time for running GASTON on our datasets was 1.17s per dataset.

Experiment 2: In Table 6.2 we give the MAP@k scores for this experiment. Table 6.4 shows the correlation matrix for the second experiment. The mean time for running GASTON on our datasets was 1.02s per dataset.

Table 6.1: The MAP@k scores for results of Experiment 1.

	MAP@1	**MAP@5**	**MAP@10**	**MAP@∞**
Compression	0.967	0.974	0.975	0.975
Frequency	0.016	0.353	0.368	0.368

Table 6.2: The MAP@k scores for Experiment 2.

	MAP@2	**MAP@5**	**MAP@10**	**MAP@∞**
Compression	0.955	0.969	0.969	0.969
Frequency	0.013	0.127	0.152	0.190

Table 6.3: Spearman correlations between several variables for Experiment 1.

	p	**Mining time**	**e**	**d**	**Mean #Nodes per comp**
AP	-0.07^{*}	-0.24^{**}	-0.23^{**}	0.12^{**}	-0.21^{**}
AP (for e > 80)	-0.14^{*}	-0.19^{**}	-0.19^{**}	0.25^{**}	-0.03
Mining Time	0.12^{**}	–	0.89^{**}	0.26^{**}	0.83^{**}

$^{**}\ p < .001$ $^{*}\ p < 0.01$

Experiment 3: Table 6.5 shows the results for the Likert values for the mined and random edit operations for the five participants of our study. Furthermore, we conduct a t-test (with and without bootstrap) and a Wilcoxon signed-rank test, to test if the mined edit operations more likely present typical edit scenarios than the random ones. The p-values are reported in Table 6.5.

Table 6.4: Spearman correlations between several variables for Experiment 2.

	p	Size at threshold	Mining time	e	Mean #Nodes per comp
AP	-0.20^{**}	0.07	-0.02	0.09^{*}	0.02
p	–	0.23^{**}	0.16^{**}	0	0.30^{**}
Size at Threshold	–	–	0.54^{**}	0.57^{**}	0.65^{**}
Mining Time	–	–	–	0.75^{**}	0.75^{**}
e	–	–	–	–	0.92^{**}

$^{**}\ p < .001$ $^{*}\ p < 0.01$

Null hypothesis H_0: *The mined edit operations do not present a more typical edit scenario than random edit operations on average.*

We set the significance level to $\alpha = 0.01$. We can see that, for all participants, the mean Likert score for the mined operations is significantly (for all statistical tests performed) higher than the mean for the random operations. Furthermore, the 95th percentile confidence interval for the difference of the means between the two groups (i.e., mined edit operations Likert values and random edit operations Likert values) is $[1.80, 2.46]$. This interval does not contain the null value. We can therefore reject the null hypothesis.

Table 6.5: Statistics for the Likert values for Experiment 3 showing that the mined change scenarios represented by the presented HENSHIN transformation rule are significantly rated more meaningful, compared to a random baseline.

Participant	Mean mined	Mean random	p-value (t-test)	p-value (Wilcoxon)	p-value (bootstrap)
P1	3.20	1.68	$11.8 \cdot 10^{-5}$	$29.0 \cdot 10^{-5}$	$11.0 \cdot 10^{-4}$
P2	4.04	2.76	$16.6 \cdot 10^{-4}$	$6.43 \cdot 10^{-3}$	$3.60 \cdot 10^{-3}$
P3	4.32	2.60	$9.30 \cdot 10^{-6}$	$5.87 \cdot 10^{-5}$	$2.00 \cdot 10^{-4}$
P4	4.32	1.08	$2.67 \cdot 10^{-15}$	$3.51 \cdot 10^{-10}$	$< 2.20 \cdot 10^{-16}$
P5	4.48	1.60	$1.17 \cdot 10^{-11}$	$1.15 \cdot 10^{-7}$	$9.00 \cdot 10^{-4}$
Total	**4.072**	**1.944**	$\mathbf{< 2.2 \cdot 10^{-16}}$	$\mathbf{< 2.2 \cdot 10^{-16}}$	$\mathbf{< 2.2 \cdot 10^{-16}}$

Note that we also correlated the frequency of the edit operations in our dataset with the average Likert value from the interviews. We found a small but insignificant correlation (Pearson correlation of 0.33, $p = 0.12$).

After their rating, when we confronted the engineers with the true results, they stated that the edit operations obtained by OCKHAM represent typical edit scenarios.

According to one of the engineers, some edit operations "can be slightly extended" (see also Section 6.4). Some edit operations found by OCKHAM, but not recognized by the participants, were identified "to be a one-off refactoring that has been performed some time ago".

In this real-world repository, we found some operations that are typical to the modeling language SysML, for example, one which is similar to the simplified operation in Figure 6.1. We also found more interesting operations, for example, the addition of `ports` with domain-specific port properties. Furthermore, we were able to detect some rather trivial changes. For example, we can see that typically more than just one `swimlane` is added to an activity, if any. We also found simple refactorings, such as renaming a package (which also leads to changing the fully qualified name of all contained elements) or also refactorings that correspond to changed conventions, for example, activities were owned by so-called `System Use Cases` before but have been moved into `Packages`.

Experiment 4: When performing the semantic lifting with the 30 selected edit operations on the 40 model differences, we first observe that we matched only 30% of the total changes in our model differences. Furthermore, with respect to our classification of the edit operations in non-critical and critical edit operations, we observe that noncritical edit operations occur more than twice as often as critical edit operations. We had 13,870 occurrences, on average, for a non-critical edit operation and 6,027 occurrences for a critical edit operation.

In Table 6.6, we report the number of changes before the semantic lifting, the number of changes after lifting (i.e., changes not covered by any change set plus the number of change sets), the number of "unlifted changes" (i.e., changes not covered by any change set), the relative compression c_{rel}, and the total compression c. Our sample of 40 model differences consists of $\sim$ 1.3 million changes. With the exception of one edit operation, we found all edit operations in our samples. On average an edit operation has 8,189 occurrences in the dataset. We report these values for all 40 model differences together (referred to as "Total" in Table 6.6), for the average over the 40 model differences (referred to as "Average" in Table 6.6), their standard deviation (referred to as "Std. Dev." in Table 6.6), the minimum, and the maximum over the 40 model differences. Furthermore, we also report the compression ratios for the case where we restricted the semantic lifting to the set of critical edit operations.

We observe a relative compression ratio c_{rel} (see definition above) of 4.00 over all 40 model differences. In contrast, the total compression ratio c is 1.29. When restricting the set of edit operations in the semantic lifting to the set of critical edit operations, we achieve a relative compression ratio c_{rel} of 5.88. Since we are considering a smaller set of edit operations for the lifting in this case, the total compression ratio for our critical edit operations is smaller.

Another observation that can be made from the semantic lifting is that the distribution of the occurrences of edit operations over the model differences are rather

Table 6.6: Statistics for the semantic lifting, showing statistic about the compression ratios achieved.

	Total Changes	Changes Lifted	Unlifted Changes	Relative Compr.	Total Compr.
Total	1,302,591	1,006,136	907,361	4.00	1.29
Total$_{Crit}$	1,302,591	1,040,814	987,194	5.88	1.25
Average	32,564.78	25,153.40	22,684.03	3.74	1.41
Std. Dev.	27,321.05	21,248.99	19,425.70	1.52	0.28
Min	16.00	10.00	0.00	1.91	1.10
Max	73,70.006	56,032.00	51,500.00	10.65	2.22

heterogeneous (see Figure 6.5), even though for some of the model differences their distribution highly correlates. We elaborate on this observation in Section 6.4.2.

6.4 Discussion

6.4.1 Research Questions

For the first research question, we wanted to investigate if OCKHAM can discover edit operations that have actually been applied to derive the histories.

RQ 1: *Is* OCKHAM *able to identify relevant edit operations in model repositories?* We can answer this question with a "yes". Experiment 1 and 2 show high MAP scores. Only for a large number of applied operations and a large size of the input graphs, OCKHAM fails in finding the applied edit operations. We can see that our compression-based approach clearly outperforms the frequency-based approach used as a baseline.

To also see whether the approach has its merits in a real-world scenario, we formulated the second research question.

RQ 2: *Is* OCKHAM *able to find typical edit operations or editing scenarios in a real-world setting?* The edit operations found by OCKHAM obtained significantly higher (mean/median) Likert scores than the random edit operations. Furthermore, we observe a mean Likert score of almost 4.1. From this we can conclude that, compared to random ones, our mined edit operations can be considered as typical edit scenarios, on average.

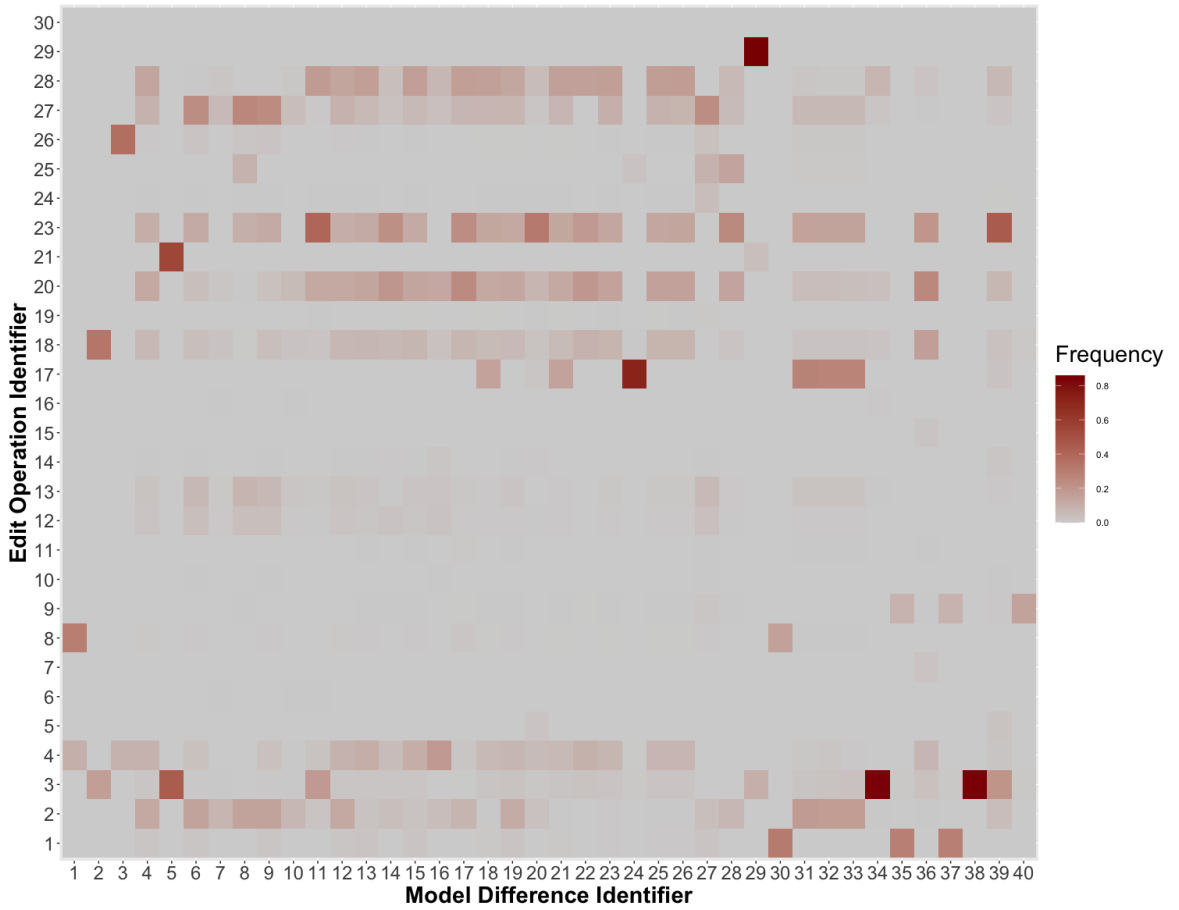

Figure 6.5: This plot shows for every model difference the distribution of the edit operations. That is, every column is a kind of change profile for the corresponding model difference. The darker the color, the higher the (relative) frequency of the edit operation in this model difference.

When looking at the mined edit operations it becomes clear, that OCKHAM is able to implicitly identify constraints, which where not made explicit in the metamodel. Also, except for one edit operation, the mined edit operations appear in the sample dataset from experiment 4. The edit operations recommended by OCKHAM are correct in most cases, and incomplete edit operations can be adjusted manually. We cannot state yet that the approach is also *complete* (i.e., is able to find all relevant edit scenarios), though.

Since we were not only interested in a binary answer, as our third research question, we asked for the main influencing drivers of the performance of OCKHAM.

RQ 3: *What are the main drivers for* OCKHAM *to succeed or fail?* From Table 6.3, we observe that increasing the number of edit operations has a negative effect on the average precision. Increasing the perturbation has a slightly negative effect,

Table 6.7: Comparison of the means for several variables of OCKHAM between the successful and failed runs in experiment 1.

	p	#Nodes per component	Size at threshold	Mining time
Overall Mean	0.55	57.6	8.20	1.26
Mean for un-detected operation	0.79	109.0	10.03	2.55

which becomes stronger for a high number of applied edit operations and therefore when huge connected (overlapping) components start to form. The number of differences d (i.e., having more examples) has a positive effect on the rank, which is rather intuitive. For the second experiment, from Table 6.4, we can observe a strong dependency of the average precision on the perturbation parameter, which is, stronger than for the first experiment. On the other hand, the correlation to the number of applied edit operations is even positive.

To analyze the main drivers further, we take a deeper look into the results. We have to distinguish between the two cases that (1) the correct edit operation is not detected at all and (2) the correct edit operation has a low rank.

Edit operation has not been detected: For the second experiment, in 22 out of 800 examples, OCKHAM was not able to detect both edit operations. In 10 of these cases the threshold has been set too high. To mitigate this problem, in a real-world setting, the threshold parameters could be manually adjusted until the results are more plausible. In the automatic approach, further metrics have to be integrated. Other factors that cause finding the correct edit operations to fail are the perturbation, average size of component, and the size at threshold, as can be seen from Table 6.7. Given a support threshold t, the *size at threshold* is the number of nodes of the t-largest component. The intuition behind this metric is the following: For the frequent subgraph miner, in order to prune the search space, a subgraph is only allowed to appear in, at most, $t-1$ components. Therefore, the subgraph miner needs to search for a subgraph, at least, in one component with size greater than the *size at threshold*. Usually, the component size plays a major role in the complexity of the subgraph mining. When the t-largest component is small, we could always use this component (or smaller ones) to guide the search through the search space and therefore we will not have a large search space. So, a large size of the component at threshold could be an indicator for a complicated dataset.

Looked deeper into the results of the datasets from the first experiment, for which the correct subgraph has not been identified, we can see that, for some of these subgraphs, there is a supergraph in our recommendations that is top-ranked. Usually this supergraph contains one or two additional nodes. Since we have a rather small meta-model, and we only use four other edit operations for the perturbation, it can

Table 6.8: Possible drivers (number of differences (d), number of applied edit operations(e), perturbation (p), mean number of nodes per component, size at threshold) for a low rank (≥ 5).

d	e	p	Mean #Nodes per component	Size at threshold	Average precision	Rank
10	92	0.3	142.2	13	0.13	8
10	67	0.4	91.0	16	0.14	7
10	78	0.8	87.3	14	0.14	7
10	98	0.8	127.7	14	0.067	15
20	81	0.1	227.0	16	0.13	8
20	99	0.1	272.2	19	0.010	99
20	100	0.1	272.7	17	0.013	78

rarely happen that these larger graphs occur with the same frequency as the actual subgraph. The correct subgraphs are then pruned away. In other words, in these cases the signal-to-noise ratio has been too low.

Edit operation has a low rank: First, note that we observe a low rank (rank ≥ 5) only very rarely. For the first experiment, it happened in 7 out of 2000 datasets, while for the second experiment, it did not happen at all. In Table 6.8, we list the corresponding datasets and the values for drivers of a low rank. One interesting observation is that, for some datasets with low-ranked correct subgraph, we can see that the correct graph appears very early in the subgraph lattice, for example, first child of the best compressing subgraph but rank 99 in the output, or first child of the second-best subgraph but rank 15 in the output. This suggests that this is more a presentation issue, which is due to the fact that we have to select a linear order of all subgraph candidates for the experiment.

In Experiment 3, we only found two mined edit operations that received an average Likert score below 3 from the five practitioners in the interviews. The first one was a refactoring that was actually performed but that targeted only a minority of all models. Only two of the participants were aware of this refactoring, and one of them did not directly recognize it due to the abstract presentation of the refactoring. The other edit operation that was not considered as a typical edit scenario was adding a kind of document to another document. This edit operation was even considered as illegal by 3 out of the 5 participants. The reason for this is the internal modeling of the relationship between the documents, which the participants were not aware of. So, it can also be attributed to the presentation of the results in terms of HENSHIN transformation rules, which require an understanding of the underlying modeling language's meta-model.

For four of the edit operations of Experiment 3, some participants mentioned that the edit operation can be extended slightly. We took a closer look at why OCKHAM was not able to detect the extended edit operation, and it turned out that it was due to our simplifications of locality relaxation and also due to the missing type hierarchies in our graphs. For example, in one edit operation, one could see that the fully qualified name (name + location in the containment hierarchy) of some nodes has been changed, but the actual change causing this name change was not visible, because it was a renaming of a package a few levels higher in the containment hierarchy that was not directly linked to our change. Another example was a "cut off" referenced element in an edit operation. The reason why this has been cut off was that the element appeared as different subclasses in the model differences and each single change alone was not frequent.

To summarize: The main drivers for OCKHAM to fail are a large average size of components and the size at threshold. The average size is related to the number of edit operations applied per model difference. In a practical scenario, huge differences can be excluded when running edit operation detection. The size of the component at threshold can be reduced by increasing the support threshold parameters of the frequent subgraph mining. With higher threshold, we increase the risk of missing some less frequent edit operations, but the reliability for detecting the correct (more frequent) operations is increased. Having more examples improves the results of OCKHAM.

In the forth research questions, we wanted to investigate the main parameters for the runtime performance of the frequent subgraph mining.

RQ 4: *What are the main parameters for the performance of the frequent subgraph mining?* From Table 6.3, we can observe a strong Spearman correlation of the mining time with the number of applied edit operations e (0.89) and implicitly also the average number of nodes per component (0.83). If we only look at edit operations with rank > 1, we observe a strong negative correlation of -0.51 of mining time and the average precision (not shown in Table 6.3). This actually means that large mining times usually come with a bad ranking. The same effect can be observed for Experiment 2 (Table 6.4). We can also see, that the mining time correlates with the size at threshold.

For the last research question, RQ 5, we took a closer look at the practical impact of the edit operations discovered by OCKHAM in the context of the case study in Chapter 4.

RQ 5: *What is the practical impact of the edit operations discovered by* OCKHAM *in the context of the case study from Chapter 4?* In Experiment 4, we lifted the changes using the edit operations discovered by OCKHAM and applied the classification into refactorings and critical edit operations. From this experiment we can see that our edit operations also occur in practice. Furthermore, the edit operations mined by OCKHAM can be used to compress model differences significantly. Since they are also meaningful to domain experts (see RQ 2), they can be used to summarize model differences. Using classifications of the edit operations, in addition, one can also use the approach to filter out irrelevant changes in the analysis of model differences.

Rather "small" changes, for example, adding a package ($\sim$ 23% of the change sets) or renaming a property ($\sim$ 10% of the change sets), occur with a much higher frequency than more complex edit operations such as adding an external port with multiplicities ($\sim$ 1.3% of the change sets). In general, from the results of Experiment 4, we can see that critical edit operations are more rare compared to refactoring edit operations. On the other hand, they provide a higher compression of the model differences. The classification of the edit operations could also be used to filter out irrelevant changes in the analysis of model differences. Nevertheless, the classification is most likely task-dependent and we cannot be sure that this observation also holds for other classifications.

Furthermore, the distribution of edit operations in a model difference can tell us something about the nature of the evolution step. For example, we can determine "hot spots", where a lot of critical edit operations are performed.

Even though we did not evaluate the semantic lifting approach in a long-term application, engineers at Siemens Mobility support the hypothesis that it can be useful for software product-line engineering related tasks and quality assurance of models (cf. Section 4.6). Overall, we can conclude that the edit operations discovered by OCKHAM can be useful in the context of the case study in Chapter 4.

To summarize the discussions of the research questions, we can clearly say that OCKHAM is able to identify relevant edit operations that correspond to typical edit scenarios in the history of projects. The mining doesn't scale to large connected model differences, so these have to be excluded from the mining. As a consequence of this, but also because a minimum frequency threshold has to be set, we can not claim that our approach is *complete* in the sense that it will discover all relevant edit operations. Furthermore, hard tasks for the subgraph miner typically lead also to bad results with respect to the correctness of the mined edit operations. The edit operations discovered by OCKHAM can also be applied in practical application scenarios, for example, to ease the analysis of large model differences by employing semantic lifting and filtering.

With regard to Hypothesis 3, we can say that the results of the experiments performed here clearly support the hypothesis that the most compressing edit operations in software model repositories are also meaningful.

6.4.2 Application: Evolution Profiles from Lifted Model Differences

We made an interesting observation when analyzing the distribution of the edit operations in the fourth experiment (see Figure 6.5). Between some sample model differences, we observe high similarities of their distributions, while for others, there is almost no correlation. In other words, we can see clusters of the edit operation distributions, which can be seen more easily in Figure 6.6.

A closer look into these clusters reveals that there is indeed a similarity in the model differences. For example, the cluster $[11-15, 17-23, 25-26]$ contains a lot of functional modifications of the models. For example, interfaces have been changed, so-called `Functional Addresses` have been added, and activity diagrams were modified. The cluster $[24, 31-33]$ also contains functional modifications, but additionally documentation has changed a lot. For example, so-called `Reference Documents` have been added, system requirements have been added or changed, and comments were added or changed. The cluster $[6-10, 27]$ contains a lot of added functionality but less modification to the documentation. Furthermore, the evolution profiles between different submodels for the same trainset type are often very similar. We can also find some model differences that are quite different from all other model differences and play some special role, for example, for cross-cutting functionality like test automation.

This suggests that the edit operations frequency distributions can be seen as a kind of *evolution profile* for the model difference, which can be used, for example, to gain further insights into the model differences. These evolution profiles can be used to reason about model differences or model evolution from a high-level perspective. Insights about the clusters can also be used, for example, to detect uncommon and possibly undesired changes. Since they are probability distributions, methods from statistics can be applied. Furthermore, they can also be seen as normed vectors (w.r.t. the $l1$-norm) and therefore also methods from linear algebra can be applied to further analyze model differences. Besides the analysis of the model evolution, the evolution profile can also be used for simulating model evolution, especially if different evolution scenarios are to be investigated. Similar profiles have already been used for model evolution simulation, for example, by Heider et al. [128], who also refer to these profiles as *evolution profiles*. Another statistical approach to model simulation based on semantic lifting has been proposed by Yazdi et al. [294] but their edit operations have to be defined manually.

Dual to this cluster analysis, the evolution profiles can also be used to analyze co-occurrences of edit operations. For example, we observe a large Pearson correlation

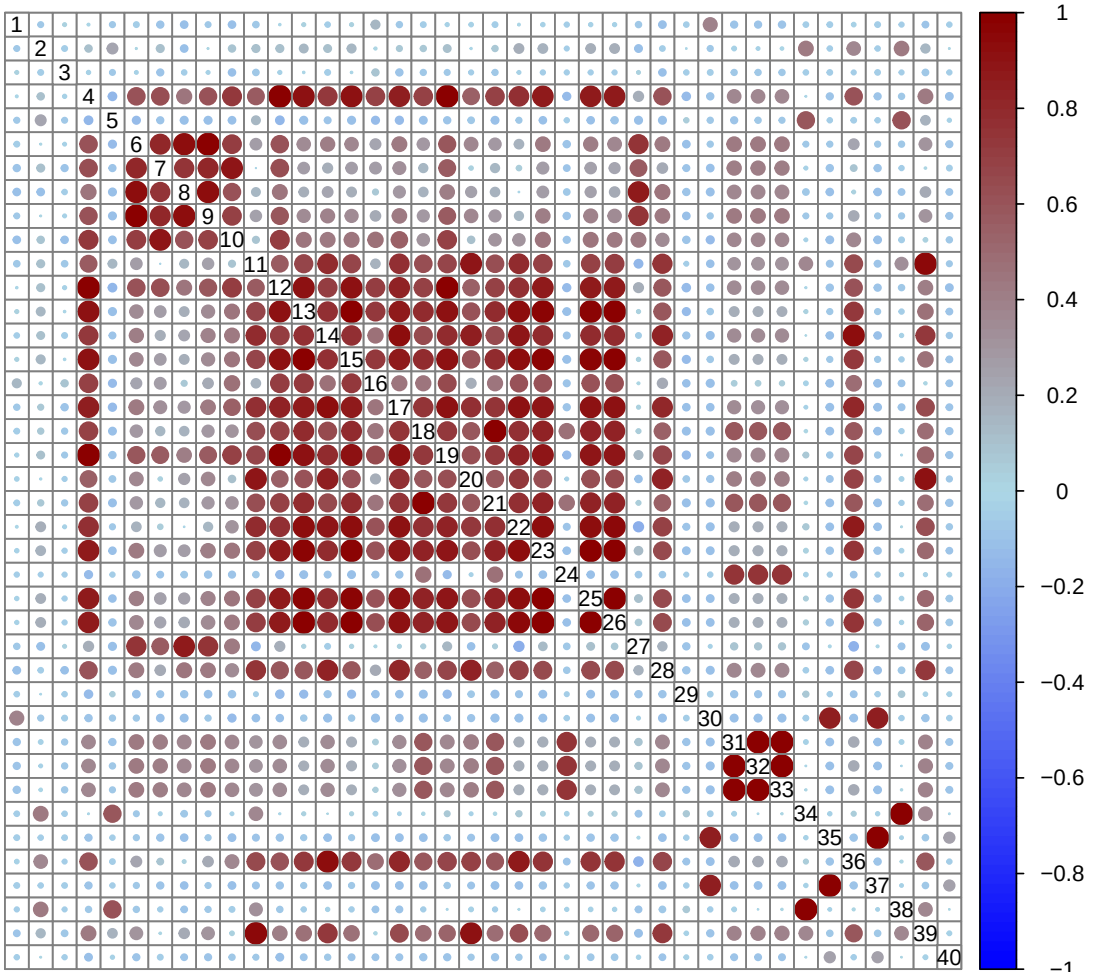

Figure 6.6: Pearson correlation of the edit operation frequency distributions for 40 sample model differences. We have removed a dominant edit operation (Adding a `Package`) from the distributions to make the clusters appear more clearly in this visual representation. Qualitatively, the same clusters will be obtained though, without removing this edit operation from the frequency distribution.

of $\sim$ 0.99 between the edit operations `ChangeUseCase` and `ChangeActivity`. This could also indicate that these edit operations can be combined to form a new, larger edit operation, which is the case in this concrete example. Some edit operations seem to be almost uncorrelated, for example, we observe that adding new functionality rarely occurs together with modifying existing functionality. Similar co-evolution analysis have been investigated in detail by Getir et al. [111]. Note that OCKHAM is also a kind of co-change analysis that also takes structural similarities into account. The interplay between statistical co-occurrence analysis and structural graph mining in the analysis of model evolution is a promising future research direction.

6.4.3 Limitations

Locality relaxation: One limitation of our approach is the locality relaxation, which limits our ability to find patterns that are scattered across more than one connected component of the simple change graph. As we have seen in our railway case study, this can lead to incomplete edit operations. Another typical example for violating the relaxation are naming conventions. As future research, one can investigate the use natural language processing techniques such as semantic matching to augment the models by further references.
No attribute information: For our experiments, we did not take attribute information into account. Attributes (e.g., the name of a component) could also be integrated into the edit operation as preconditions or to extract the parameters of an edit operation. For the purpose of summarizing a model difference or identifying violations in a model difference, preconditions and parameters are not important and only the presence of structural patterns matters. We later come back to this shortcoming in our experiments on model completion (cf. Chapter 8), where we employ language models that will also process attribute information.
Application to simplified graphs: Generally, an edit operation is a model transformation. Model transformation engines such as HENSHIN provide features to deal with class inheritance or multi-object structures (roughly speaking, foreach loops in model transformations). In our approach, we are not leveraging these features yet. They could be integrated into OCKHAM in a post-processing step. For example, one possibility would be to feed the example instances of patterns discovered by OCKHAM into a traditional model-transformation-by-example approach [160].
Transient effects: We do not take so-called transient effects into account yet. One applied edit operation can invalidate the pre- or post-conditions of another edit operation. However, we have seen in our experiments that this only causes problems in cases where we apply only a few "correct" edit operations with high perturbation. In a practical scenario, the "perturbations" will more likely cancel each other out. When a transient effect occurs very frequently, a new pattern will be discovered. That is, when two (or more) operations are always applied together, we want to find the composite pattern, not the constituent ones.
Focus on single subgraphs instead of sets: Another limitation is the fact that we focused the optimization on *single* edit operations but not a *complete set* of edit operations. One could detect only the most-compressing edit operation and then substitute this in the model differences and re-run the mining to discover the second most-compressing edit operation and so on. Another solution would be to detect a set of candidate edit operations using OCKHAM and then select an optimal set using a meta-heuristic search algorithm optimizing the *total compression*. We leave this for further research. As already mentioned, more recent compression-based graph miners such as GRAPHMDL [28] would be a good starting point.

Completeness of the set of mined edit operations: As we can see from Experiment 4, a set of 30 edit operations is still far from being complete. In fact, with this set, we missed 70% of changes in our sample model differences. Anyway, the approach can also be applied iteratively, in the sense that edit operations which are already approved by domain experts can be used to substitute the corresponding subgraph by a single node. The graph mining can then be applied on these *contracted* graph databases until all changes are included in a change set. However, since a change does not unambiguously belong to a change set, this approach would be subject to the limitation from the previous paragraph.

6.4.4 Threats to Validity

Internal validity: We have designed the first two experiments such that we can control input parameters of interest and observe their effect on the outcome. OCKHAM makes assumptions such as the locality relaxation, which could impair real-world applicability. Because of this and since we can not claim that the results from the first two experiments also hold true in a real-world setting, we additionally applied OCKHAM to an industrial case study. Our results increase our confidence that OCKHAM also gives reasonable results in a practical scenario.

In our simulations, we applied the edit operation randomly to a meta-model. To reduce the risk of observations that are only a result of this sampling, we created many example models. In the real-world setting, we compared the mined edit operations to random ones to rule out "patternicity" [298] as an explanation for high Likert rankings. None of our participants reported problems in understanding HENSHIN's visual notation, which gives us confidence regarding their judgements (despite for misconceptions). The participants of the interviews in the third experiment were also involved in the project where the model history was taken from. There might be the risk that the interviewees have only discovered operations they have "invented". In any case, because of the huge project size and because 22 out of 25 of the edit operations were recognized as typical by more than one of the participants, this is unlikely.

External validity: Some observations in our experiments could be due to the concrete set of edit operations in the example or even due to something in the meta-models. In the future, OCKHAM has to be tested for further meta-models to increase the external validity of our results. We have validated our approach in a real-world setting, which increases our confidence in its practicality, though. As can be seen from Experiment 4, with the set of 30 edit operations, 70% of the changes are not part of a change set. It is therefore not yet clear, if the results regarding the compression of model differences also hold true when using larger sets of *meaningful* edit operations. Since we have

used an exact subgraph miner, we can be sure that the discovered edit operation are independent of the subgraph mining algorithm.

6.5 Related Work

Various approaches have been proposed to (semi-)automatically learn model transformations in the field of model transformation by example . In the first systematic approach of model transformation by example, Varro et al. [329] proposes an iterative procedure that attempts to derive *exogenous* (i.e., source and target meta-model are different) model transformations by examples. Appropriate examples need to be provided for the algorithm to work. Many approaches to learning exogenous model transformations have been proposed until now. For example, Berramla et al. [36] use statistical machine translation and language models to derive transformations. Baki et al. [26] apply simulated annealing to learn operations. Regarding *exogenous* transformations there is also an approach by Saada et al. [278], which uses graph mining techniques to learn concepts, which are then used to identify new transformation patterns. As mentioned in the introduction, most closely related approach to ours is model transformation by example for *endogenous* model transformations. Compared to exogenous model transformation by example, there are only a few studies available for endogenous model transformation by example. Langer et al. [41] present a tool called OPERATION RECORDER, which is a semi-automatic approach to derive model transformations by recording all transformation steps. A similar approach is presented by Gray et al. [315], who also infer complex model transformations from a demonstration. Jalali et al. [15] learn transformation rules from a set of examples by generalizing over pre- and postcondition graphs. Their approach has been applied to the derivation of edit operations, including negative application conditions and multi-object patterns by Kehrer et al. [160]. Instead of learning a single operation, Mokaddem et al. [234] use a genetic algorithm to learn a set of refactoring rule pairs of examples before and after applying refactorings. The creation of candidate transformations that conform to the meta-model relies on a "fragment type graph", which allows them to grow candidate patterns that conform to the meta-model. Their algorithm optimizes a model modification and preservation score. Ghannem et al. [112] also use a genetic algorithm (i.e., NSGA-II) to learn model refactorings from a set of "bad designed" and "good designed" models. Their approach distinguishes between structural similarity and semantic similarity and tries to minimize structural and semantic similarity between the initial model and the bad designed models and to maximize the similarity between the initial and the well-designed models.

All of these approaches for learning endogenous model transformations are (semi-)supervised. Either a concrete example is given (which only contains the transformation to be learned) or a set of positive and negative examples is given. In the case of Mokaddem et al.'s genetic approach, it is assumed that all transformations that can

be applied are actually applied to the source models. For the meta-model used in our real-world case study, we do not have any labeled data. In general, we are not aware of any fully unsupervised approach to learn endogenous model transformations. To reduce the search space, we leverage the evolution of the models in the model repository, though. We do not directly work on the models as in the approaches discussed above, but we work on structural model differences.

Regarding one of our motivations for mining edit operations, namely to simplify differences, there are several approaches in the source code domain [215, 359]. These approaches are more comparable to the approach of semantic lifting [163], to aggregate or filter model differences according to given patterns but they are not learning the patterns themselves. There are also approaches to mine change patterns in source code. For example, Dagit et al. [74] propose an approach based on the abstract syntax tree, and Nguyen et al. [241] mine patterns based on a so-called fine-grained program dependence graph. Janke et al. [145] derive *fine-grained edit scripts* from abstract syntax tree differences and transform them to a graph representation for graph mining. There is also some work that focuses on mining design patterns from source code [27, 84, 97, 249]. The idea behind these approaches—learning (change) patterns from a version history—is comparable to ours. In contrast to these approaches, OCKHAM works on a kind of abstract syntax graph, which already includes domain knowledge given by the meta-model. Furthermore, we do not use a similarity metric to detect change groups or frequent changes but use an (exact) subgraph mining approach. In Model-driven Engineering, one often has some kind of identifiers for the model elements, which makes the differencing more reliable and removes the need for similarity-based differencing methods.

6.6 Conclusion

We have proposed an approach, OCKHAM, for automatically deriving edit operations specified as in-place model transformations from model repositories. OCKAHM is based on the idea that a meaningful edit operation will be one that provides a good compression for the model differences (see Hypothesis 3). In particular, it uses frequent subgraph mining on labeled graph representation of model differences to discover frequent patterns in the model differences. The patterns are then filtered and ranked based on a compression metric to obtain a list of recommendations for meaningful edit operations. To the best of our knowledge, OCKHAM is the first approach for learning domain-specific edit operations in a fully *unsupervised* manner, that is, without relying on any manual intervention or input from a developer or domain expert.

We have successfully evaluated OCKHAM in two controlled experiments using synthetic ground-truth EMF models and on a large-scale real-world case study in the railway domain. We found that OCKHAM is able to extract edit operations that

have actually been applied before and that it discovers meaningful edit operations in a real-world setting. In fact, OCKHAM and the experiments performed in this chapter can be considered to be an empirical evidence for Hypothesis 3 formulated in Chapter 3.

We also found some limitation of the concrete approach presented in this chapter: Including too large components in the difference graphs can adversely affect OCKHAM in discovering the applied edit operations. Performance mostly depends on the number of applied edit operations in a model difference. Furthermore, we have shown how the edit operations discovered by OCKHAM can be used in practical applications. For example, they can be used in a semantic lifting, to express model differences in terms of edit operations and therefore "compress" the model differences. Depending on the concrete task related to the model differences, the lifted model differences can then also be filtered for the edit operations of interest. The frequency distributions of edit operations in model differences can also be used for a high-level analysis of the model differences, for example, to discover hot spots, where a lot of functional evolution has happened. OCKHAM can be applied to models of any Domain-Specific Modeling Language for which model histories are available. New effective edit operations that are performed by the users can be learned at runtime and recommendations can be made.

7

Pattern Memorization in Generative Models

If you understand something in only one way, then you don't really understand it at all. The secret of what anything means to us depends on how we've connected it to all other things we know. Well-connected representations let you turn ideas around in your mind, to envision things from many perspectives until you find one that works for you.

— Marvin Minsky

This chapter shares material with a preprint published on arXiv [321] titled "Towards Automatic Support of Software Model Evolution with Large Language Models" and the Master Thesis by Alisa Welter [343] with the title "Towards Model Editing Pattern Detection via Graph Variational Autoencoders".

This chapter dives into the application of parametric generative models, specifically large language models, and graph neural networks, for mining edit operations from software models. The theory behind the approach, and the connection of edit operation to generative models has been presented in Chapter 3 of Part I of this thesis. Inference of edit operations from model histories is based on the belief that histories reflect the elementary building blocks and patterns that modelers use to evolve their models. The focus of this chapter is on the evaluation of the ideas and a concrete application of available technologies. Furthermore, we want to understand whether patterns that exist in the data will be stored in a neural network model's weights and biases.

In practice, patterns have to be applied in a very specific context, for example, with adjusted parameters, naming conventions, etc. Graph mining approaches—as

the one presented in Chapter 6—have the limitation that it is hard to capture these context-specific patterns, without losing the generalization power of the approach. Without making any domain-specific assumptions, it is hard to capture semantics of a pattern that goes far beyond a pure type level. For many real-world languages, large parts of the semantics are not captured by the type system. To overcome this challenge, we will therefore investigate parametric machine learning models (cf. Section 3.4) as a promising technology to capture semantics from model histories. The analysis and experiments presented in this chapter will be a first step towards validating the hypothesis that parametric generative models are a feasible approach to model software model evolution (see Hypothesis 2).

With this chapter, we make the following contributions:

- We propose an approach to use large language models for mining edit operations.
- We evaluate the approach in a controlled experiment. We find that the approach can generate correct edit operations.
- We investigate whether patterns in the model histories will be encoded in a machine learning model's parameters, when training a generative model on the model histories. We find initial evidence that the patterns have their representation in the models parameters.

7.1 The Long Tail of Domain-specific Knowledge

In Section 6.4, we have discussed limitations of OCKHAM. In order to better understand if and how these limitations can be overcome, we will study some alternative approaches in the present chapter. Still, the main spirit behind the approaches that we want to investigate are similar to OCKHAM: We believe that inference of meaningful edit operations (see Definition 3.5.1), can be done by leveraging historic data (see Hypothesis 1).

7.1.1 Shortcomings of the "Symbolic" Pattern Mining

Because of OCKHAM relies on frequent subgraph mining, it faces scalability issues, particularly with large model differences, leading to long execution times and high memory consumption. In particular, there is a trade-off between the completeness of the mined edit operations and the computational complexity (i.e., low thresholds

for the minimum support lead to a high number of edit operations but also to long execution times). Indeed, extending OCKHAM also to related but not directly connected patterns would lead to too large graphs that, in most cases, can not be handled by exact frequent subgraph mining. Additionally, it lacks abstraction capabilities, making it difficult to generalize edit operations that differ only slightly.

Example Long Tail of Domain-Specific Knowledge.
Consider the following scenario: In a production system, the length and speed of a conveyor belt have been increased. This, in turn, leads to adjustments of the sizing of the motor. From the change of the speed and length of the conveyor belt, the electrical engineer would typically know how to adjust the dimensions of the motor. Anyway, the electrical engineering and automation tools typically would not include the complex change, including all the necessary adjustments. Indeed, the tool providers could include the calculations of the correct sizing and include necessary edit operation in the tool. That is, based on independent variables (length and speed of the conveyor belt), dependent variables (e.g., correct motor, power supply, controller settings, signal configuration, etc.) would automatically be computed by the electrical and automation engineering tool.
Taking this idea further, imagine one would provide edit operations for each possible change scenario. For many real-world, complex domains, the tool would need to provide a vast and incomprehensible amount of edit operations. In particular, for complex domains it is unlikely that every change scenario can be anticipated in advance.

For many domains, having a complete set of edit operations and adaptation logic for any future situation, is not feasible. This can be seen as a consequence of a *long tail* of domain-specific knowledge. Explicitly encoding all necessary knowledge would require a lot of manual upfront work. Also, the patterns would evolve and one would even have to model a moving target. Besides, some rules are even not explicitly known (i.e., they are tacit knowledge). OCKHAM is able to capture some domain-specific knowledge, but it can not capture semantics that is not encoded in the structure of the differences and labels. The idea to infer programs from examples or declarative constraints has been studied in program synthesis research [120, 261]. One approach to extend OCKHAM to richer semantics, would be to combine graph mining with existing program synthesis approaches. Anyway, also these approaches are limited to an underlying grammar and rarely can capture the long tail of domain-specific knowledge.

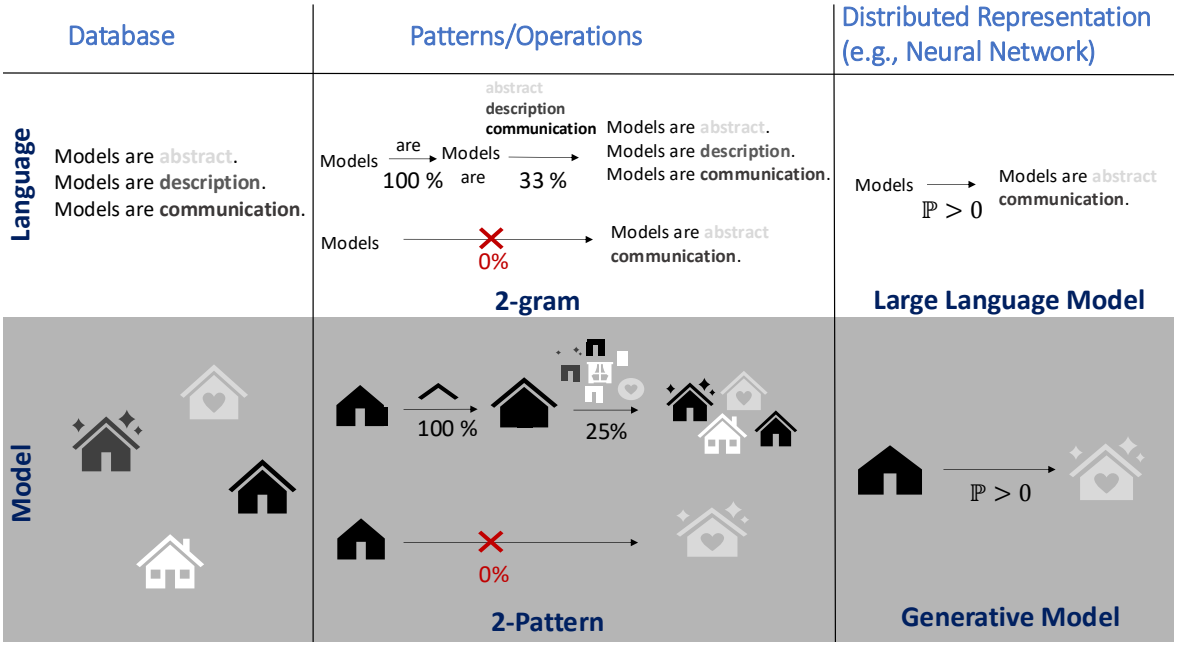

Figure 7.1: Similar to large language models, distributed representations can be used to solve an "out-of-history" challenge for context-specific model generation.

7.1.2 Solution Idea: Distributed Representations

Similar to natural language processing, the idea to overcome this long tail challenge of domain-specific knowledge might be to use distributed representations (cf. Section 3.5.4). As suggested by Figure 7.1 a proper generative model would assign a probability greater than zero to unseen (but realistic) edit scenarios.

Motivated by the success of large language models and graph neural networks in graph tasks [25, 87, 196], we investigate if these models can be used to generate and mine edit operations from software model histories. As we discussed in Section 3.5.4, the idea is to train a parametric machine learning model to approximate a distribution behind the input dataset. Applied to software model repositories as training data, similar to OCKHAM (cf. Chapter 6), generative machine learning models represent a compression[1] of the model repositories, because the generative machine learning model can then be used to approximately generate the software model repositories. Therefore, a natural question to ask is whether these machine learning models store information about typical editing patterns (i.e., edit operations) in their parameters.

In this chapter, we investigate if and to what extent edit operations patterns in the input data are stored in the machine learning models. In particular, we will postpone the solution of the challenges mentioned above to Chapter 8. We will then investigate

1 We assume that the data in the repositories is larger than the data that can be stored in the neural network model's weights and biases.

several techniques, such as *probing*, to determine whether pattern representation are present in the trained model's parameters.

In conclusion, we find that for both—large language models and graph neural networks—the models seem to memorize patterns in the data.

7.2 Methodology

The goal of this chapter is to understand if, in principle, large language models and graph neural networks are suitable technologies for learning edit patterns from edit histories stored in model repositories.

In Section 3.5.4 and in Example 3.4.1, we argued that software modeling can be described as a generative model (see Definition 3.4.1). We also argued in favor of parametric neural network machine learning models, to support the modeling process. Regarding the goal of this chapter, this leads to a new challenge: The patterns—such as edit operations—are not explicitly available in these models. In this section, we will therefore describe several approaches—depending on the technology—to investigate whether patterns have been "learned" by the neural network approach.

In particular, there are two[2] *strategies* of how we could investigate the "pattern knowledge" of the model: We could either try to use the generative model to generate (or read out) the patterns, or we could use embeddings (i.e., activations in the neural network layers for some given input) for a "downstream task" that would require knowledge about the patterns—a technique called *probing*.

We describe the approaches in detail for each individual technology.[3]

7.2.1 Graph Neural Network

As the architecture for a generative graph neural network approach, we choose the Graph Variational Autoencoder from Definition 3.4.2 (cf. Section 3.4.1).

In Section 3.4.1, we only presented the general idea of the Graph Variational Autoencoder. Still, the concrete realization of the encoder and the decoder, as well as the concrete approximation of the reconstruction error in the loss function, are not

2 A third strategy, which we did not investigate as part of this thesis, would be to estimate the probability of a pattern using the generative model. One would then expect that the probability of the pattern is typically higher compared to arbitrary structures, that is, that the *surprisal* is low.

3 The reason to use different approaches is mainly due to availability and cost. At the time the experiments with large language models were conducted, we had no access to the embeddings of a fine-tuned model for the state-of-the art models (i.e., GPT-3). On the other hand, the approach for pattern generation that we employ for large language models is only available for autoregressive models (i.e., an iterative generation of tokens depends on the previous generation). The graph neural network architecture that we will investigate, unfortunately, is not autoregressive.

specified. As a starting point for our implementation, we used an implementation based on the work of Kwon et al. [196].[4]

We chose this implementation due to its availability and simplicity. Their approach uses the following approximate reconstruction loss function:

Definition 7.2.1 Reconstruction Loss of the Graph Variational Autoencoder. *Let $\mathbf{v}_i$ and $\mathbf{e}^{i,j}$ be the entries of the node and edge embeddings of the input graph, respectively, and $\widetilde{\mathbf{v}}_i$ and $\widetilde{\mathbf{e}}^{i,j}$ the node and edge embeddings of the reconstructed graph, and let b be the edge label. The reconstruction loss of the graph variational autoencoder [196] is then given by:*

$$\begin{aligned}
&\mathbb{E}_{\mathbf{z}\sim q_\theta(\mathbf{z}|x)}\left[-\log p_\phi(x \mid \mathbf{z})\right] \simeq \left\| \left(\sum_i \mathbf{v}_i\right) - \left(\sum_i \widetilde{\mathbf{v}}_i\right) \right\|^2 \\
&+ \left\| \left(\sum_{i,j} \mathbf{e}^{i,j}\right) - \left(\sum_{i,j} \tilde{\mathbf{e}}^{i,j}\right) \right\|^2 \\
&+ \sum_b \left\| \left(\sum_{i,j} e^{i,j,b}\mathbf{v}^i\right) - \left(\sum_{i,j} \tilde{e}^{i,j,b}\widetilde{\mathbf{v}}^i\right) \right\|^2 + \left\| \left(\sum_{i,j} e^{j,i,b}\mathbf{v}^i\right) - \left(\sum_{i,j} \tilde{e}^{j,i,b}\widetilde{\mathbf{v}}^i\right) \right\|^2 \\
&+ \sum_{\mathrm{b}} \left\| \left(\sum_{i,j} e^{i,j,b}\mathbf{v}^i\mathbf{v}^{jT}\right) - \left(\sum_{i,j} \tilde{e}^{i,j,b}\widetilde{\mathbf{v}}^i\widetilde{\mathbf{v}}^{jT}\right) \right\|^2
\end{aligned}$$

This loss consists of five terms, which are the squared differences of the node embeddings (i.e., measures if the correct nodes are in the reconstructed graph), the edge embeddings (i.e., measures if the correct edges are in the reconstructed graph), incoming and outgoing edge-node pairs, and node-edge-node triples (i.e., measures if the correct pair-wise node connections are in the reconstructed graph). That is the main idea is to approximate the distance between the original graph } and the reconstructed graph $\widetilde{g}$ by comparing the distribution of node labels, edge labels, node-edge label combinations, and node-edge-node label combinations.

In our experiments, we will vary some graph neural network elements (e.g., the convolutional layer architectures), optimize some others via a hyperparameter optimization (e.g., the number of layers, layer size, or the learning rate), and we will also evaluate the impact of the loss function on the results. We will also fix some aspects of the approach (e.g., the aggregation function in the convolutional layers), in order to keep the combinatorial complexity manageable.

The Graph Variational Autoencoder will then be trained on a dataset of simple change graphs. That is, the training task for this network is to reconstruct simple change graphs.

4 https://github.com/deepfindr/gvae

Generation of simple change graphs: After training, we can use the decoder $p_\theta(\mathbf{X} \mid \mathbf{Z})$ to generate simple change graphs from the latent space. In our concrete implementation, we will generate simple change graphs by normally distributed sampling from the latent space and then computing the graph representation X from z and p_θ. The idea is that the generated graphs resemble the input simple change graphs. Unfortunately, there is no "straightforward" way to measure the similarity of the generated simple change graphs (cf. Section 3.3.2). For this work, we therefore compare distributions of several graph invariants (e.g., the number of edges, the number of nodes, or the degree distribution) between the generated and the input simple change graphs.

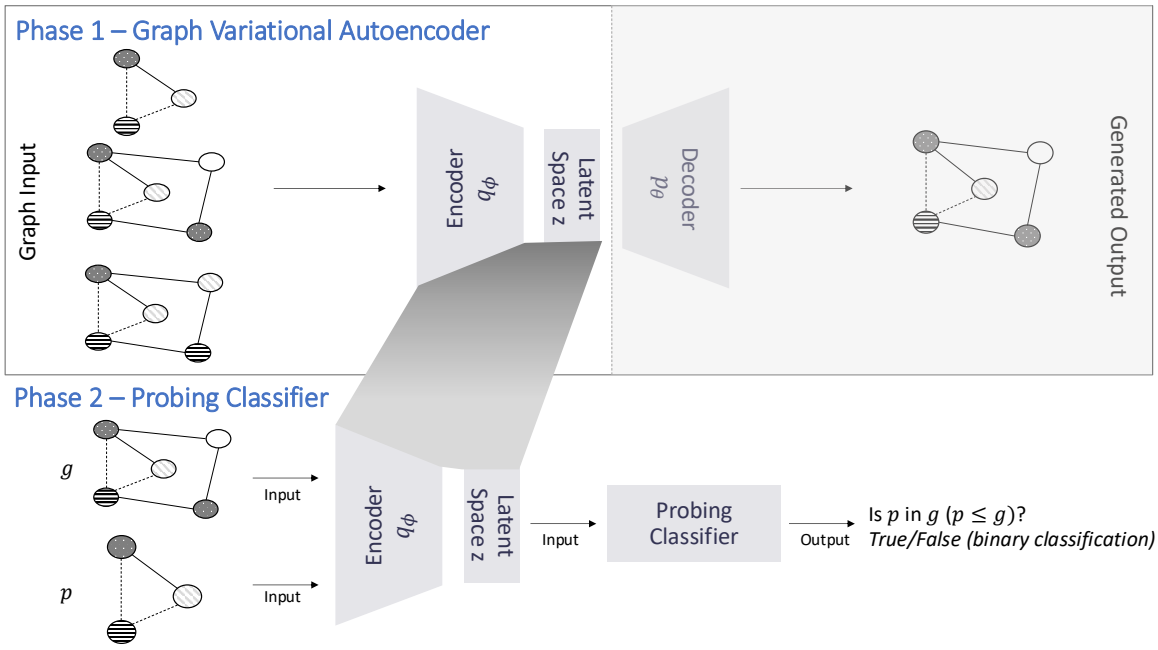

Figure 7.2: A schematic description of the probing classifier. After a Variational Autoencoder has been trained, the latent representation is reused as input for a classifier. The task for this classifier is to determine if a certain pattern p is contained in an input graph g.

Probing (see Figure 7.2): There are two probing methods we use in our experiments: First, we can train a binary classifier that, given an encoding z of simple change graph representation, predicts the presence of a specific pattern. Second, we can train a regressor that predicts the frequency of a specific pattern in the simple change graph representation.

For both methods, we train a simple neural network on the latent space representation on a part of the input dataset that we held-out from the training of the Graph Variational Autoencoder.

To ensure a robust training process for the classifier, we have to address imbalanced datasets, by employing *oversampling*. Alternative resampling techniques, such as

undersampling or the creation of synthetic instances of the minority class, were considered but ultimately deemed infeasible due to the exceedingly low frequencies of certain classes.

7.2.2 Large Language Model

Alternatively to graph neural networks, we will also study large language models for edit operation mining.

Motivation
Many frequent (or compressing) subgraph miners [72, 170, 355], as well as OCKHAM, grow the patterns (also called motifs) edge-wise. That is, they start with a single node and extend it edge by edge. Given a context graph G, two possible extensions edges e and e', and some metric M (e.g., frequency or compression), the search then favors the extension e with better metric $M(e \cap g) > M(e' \cap g)$ (e.g., in the form of a beam search as in Subdue [170]). For a frequency and a compression measure, this condition can then be reformulated to yield a probabilistic formulation:

$$
\begin{aligned}
& M(e \cap G) > M(e' \cap G) && (7.1)\\
\Leftrightarrow \quad & \mathbb{P}(e \cap G) > \mathbb{P}(e' \cap G) && (7.2)\\
\Leftrightarrow \quad & \frac{\mathbb{P}(e \cap G)}{\mathbb{P}(G)} > \frac{\mathbb{P}(e' \cap G)}{\mathbb{P}(G)} && (7.3)\\
\Leftrightarrow \quad & \mathbb{P}(e \mid G) > \mathbb{P}(e' \mid G) && (7.4)
\end{aligned}
$$

In (7.2), we have normalized over the whole dataset to yield a probabilistic formulation and in (7.4), we apply the definition of the conditional probability. In this formulation, the extension criteria reminds a lot on the language models from Section 3.4.2, with the major difference that language models are probability distributions on sequences of tokens, while the formulation above is for a probability distribution on sets of graphs. This suggests that language models can be used to generate serializations of patterns in simple change graphs in an edge-wise fashion.

Concrete Approach
The motivation above suggests the following high-level procedure to employ language models for the mining of edit operations:

Step 1: Serialization: We serialize simple change graphs of (successive) pairs of software models edge-wise.

Step 2: Training Data Generation: We then generate pairs of partial simple change graphs and their completion to the full simple change graph serialization.

Step 3: Fine-Tuning: These pairs are then used to fine-tune a pre-trained language model. As base pre-trained language models, we use the models text-ada-001, text-curie-001, and text-davinci-003 from the GPT-3 family of language models.

Step 4: Candidate Generation: The fine-tuned language model can then be used to generate serializations of simple change graphs. Note that, when a context is already given, we are in a model completion setting.

Step 5: Re-Ranking: In a last step, we rank the generated serializations from the previous step.

The approach is therefore divided into two phases: A *training phase*, in which a language model is fine-tuned on the serialization of simple change graphs (Step 1 – Step 3), and a *generation phase* (Step 4 and Step 5), in which the fine-tuned language model is used to generate serializations of simple change graphs (which we hope to coincide with meaningful edit operations). We describe the approach in detail in the appendix in Section D.2.

Of course, other approaches would be conceivable. For example, one could sample simple change graphs from the dataset and then apply the candidate generation algorithm with the samples in the context. One could also use a chat-like interface [250] and, given a set of sample simple change graphs, instruct the model to output some edit operation candidates. To keep things simple and avoid combinatorial explosion in our experiments, we fixed the edit operation mining algorithm to the one described above.[5]

7.3 Evaluation

In this section we will formulate and answer research questions to better understand if and to what extent generative neural network models capture pattern knowledge.

The goal of this chapter is twofold. First, we want to understand whether pattern knowledge is encoded in the generative neural networks that we use for our approach. Second, in the concrete example of large language models, we want to understand if we can also read out the pattern knowledge from the large language model.

5 Furthermore, at the time of our initial experiments performed in this area, reasonably powerful chat-like models were not available. Anyway, in Chapter 8, in the more general case of model autocompletion, we will investigate also chat-like language models.

7.3.1 Research Questions for Experimental Context

To achieve our research goal we formulate separate research questions for graph neural networks and large language models, accounting for the different techniques (cf. Section 7.2) that we apply in the two cases.

Graph Neural Networks
Regarding graph neural networks, we will investigate how well generated graphs resemble the input simple change graphs. Given a certain simple change graph as input to the graph neural network, we also want to predict whether it contains a pattern subgraph from its embedding—or even count the number of pattern occurrences.

We therefore ask the following research questions:

> **RQ 1:** *How well can the proposed Graph Variational Autoencoder approach resemble the original simple change graph distribution?*

By "resembling" the original simple change graph distribution, we mean that the generated graphs should have similar distribution of several graph properties. Furthermore, the generated graphs should be correct according to the meta-model. To answer this research question, we also want to know how the performance of the Graph Variational Autoencoder depends on the architecture of the encoder layer.

> **RQ 2:** *Does the proposed Graph Variational Autoencoder approach "memorize" edit operation that have been used to construct the training data?*

Given an edit operation and a simple change graph as input, can we identify whether and how often this edit operation has been applied by solely relying on the embeddings of the simple change graph? Applying this so-called *probing*, we can analyze features that have been learned by the graph neural network. Drawing inspiration from existing work (e.g., the work by Liu et al. [208]), we aim to use probing to discover pattern knowledge encoded implicitly in the neural network's weights and biases. To answer this research question, we also want to understand how the performance of the ability to identify edit operation depends on the architecture and hyperparameters of the graph neural network.

Large Language Models
To better understand the merits of language models for edit operation mining, we want to generate (or "read-out") the pattern using the approach presented in Section 7.2. We will then evaluate the generated edit operation candidates with respect to the properties of the dataset and the properties of the language model used for the approach.

RQ 3: *Using our approach, are language models capable of generating correct simple change graph graph serializations?*

Since a language model is not aware of the meta-model and definition of a graph *per se*, the generated edit operations might not be correct simple change graph serializations. That is, they might be invalid according to the meta-model (e.g., invalid combination of edge, source, and target node labels) or could even be invalid directed labeled graph serializations (i.e., not adhere to the given serialization format, namely the EdgeList format from Section D.2 for our experiments). We are particularly interested how this depends on the properties of the dataset and the properties of the language model used for the approach.

RQ 4: *Using our approach, are language models capable of providing correct auto-completions for software models?*

The approach uses a language model that is trained to complete the simple change graph serializations. It optimizes the *token probability*, given a context. This does not ensure *per se* that the provided completions represent simple change graphs that are isomorphic to the *correct* simple change graphs. Therefore, we are interested in to which extent the completed and the original simple change graph coincide and how this depends on the properties of the dataset and of the language model.

RQ 5: *Using our approach, can edit operations be reconstructed from the language model?*

The main idea of our approach is that the language model leverages patterns in the training data while generating text. Therefore, it should be possible to read out the patterns from the language model. The approach presented here is just one idea how the patterns can be retrieved from the language model, and we have to evaluate this approach empirically. We further evaluate how the generation of edit operation candidates depends on the properties of the dataset and the properties of the language model used for the approach.

Furthermore, we are interested in the ranking of the generated edit operation candidates: There are several possibilities to rank the list of generated edit operation candidates, including language model probability, the probability scaled by the factorial of the number of edges, or scaled by the number of edges, or a more computationally expensive compression-like metric as used by OCKHAM from Chapter 6. The idea behind the scaled metrics is that—as discussed already above—there are several possible simple change graph serialization (up to $|E|!$). At least, the probability will inevitably decrease with the number of edges, and we have to account for this to avoid favoring smaller edit operation candidates.

7.3.2 Dataset

In this chapter, we will only work with the SYNTHETIC dataset (cf. Chapter 5). The reason is that we need a ground truth of edit operations for our experiments. Furthermore, this chapter can rather be seen as a proof of existence. In Chapter 8 we will then conduct our experiments on all datasets, that is, INDUSTRY, REPAIRVISION, and SYNTHETIC dataset.

Graph Neural Networks
To answer RQ 1, we use a subset of the SYNTHETIC dataset. Table 7.1 lists the concrete selection.

Table 7.1: Subset of the SYNTHETIC dataset used for the graph generation experiment.

Dataset name	Description	# Graphs	# Pattern
Dataset Small 1	Random selection of graphs	780	832
Dataset Small 2	Random selection of graphs	1583	1991
Dataset Small 3	Random selection of graphs	2048	2353
Dataset Small Small-E-values	Focus on model histories with small e values ($e \in \{1, 11\}$)	952	1140
Dataset Small Small-P-values	Focus on model histories with small e values ($p = 0.0$)	2451	2609
Dataset Small Large-E-values	Focus on model histories with large e values ($e = 81$)	4807	5897
Dataset Small Large-P-values	Focus on model histories with large p values ($p = 1.0$)	3918	5385
Dataset Large	Large data amount ($e \in \{21, 31, 41, 51, 61, 71\}$)	7801	9427

Furthermore, because we have a fixed size input for the graph neural network, we have to decide on a fixed size for the simple change graphs. Based on the distribution of the number of nodes in the SYNTHETIC dataset, we decide to use a fixed size of 10 nodes (for one graph component), which still covers the majority of the graphs in the dataset (because the simple change graphs are small compared to the size of the abstract syntax graph).

Regarding RQ 2, unfortunately, the SYNTHETIC dataset is not ideal. In the generation of this dataset (cf. Chapter 5), the ground truth edit operations have been applied to almost every revision. Consequently, train and test datasets for the probing classifier would be highly imbalanced. We therefore have to take a different avenue. We employ a graph generator [208] that has specifically been developed to evaluate counting subgraph isomorphisms. The generator combines the pattern subgraph via arbitrary partial graphs, ensuring that the combination does not lead to new

occurrences of the pattern subgraphs. For our project, we have generated graphs mimicking the characteristics of the SYNTHETIC dataset. The patterns used in the generation of the graphs including patterns subgraphs with similar characteristics to the ground truth edit operations from the SYNTHETIC dataset. By "similar," we refer to graph properties such as node and edge distributions, as well as the average count of incoming and outgoing edges. The generated dataset will then also include the ground truth values, that is, the frequency of a certain pattern in the graph.

Large Language Models
To reduce costs in the fine-tuning, we apply the approach only to a subset of the datasets from the SYNTHETIC dataset. Concretely, we only select the simulated repositories with the number of applied edit operations $e \in \{11, 31, 51, 81\}$ and the perturbation parameter $p \in \{0.0, 0.5, 1.0\}$. In total, this leads to 24 (out of originally 2000) repositories.

7.3.3 Operationalization

In this section, we describe the concrete experiments designed to answer the research questions from Section 7.3.1.

Graph Neural Networks

To answer RQ 1 and RQ 2, we will conduct two experiments. Additionally, we performed a pilot experiment to debug the initial architecture and determine reasonable hyperparameters that we do not want to control for in the subsequent experiments.

Experiment 1: The purpose of this experiment is to understand how well the proposed Graph Variational Autoencoder approach can resemble the original simple change graph distribution.

In this experiment, we keep certain variables constant. For example, the aggregation function is based on the choice of the layer architecture and not directly controlled as part of the experiment. As the readout function, we use a fixed global pooling. Given a graph g, as the *graph-level readout* (see Definition 3.3.4) we use the sum of the node embeddings of the graph:

$$r_g = \sum x_n, \tag{7.5}$$

where x_n is the node embedding of node n.

The reason to fix these variables is two-fold: First, we want to keep the combinatorial complexity low, and, second, there are studies dedicated to the influence of these variables on classification or regression tasks [314].

What we control for in this experiment are the convolutional layers. We use three different types of convolutional layers:

- TransformerConv [300] (TRF),
- GINEConv [136] (GIN),
- GATConv [331] (GAT).

The reason to choose these three types of convolutional layers is that they are thought to be strong in learning global graph structure [354].

We also perform a hyperparameter optimization on one of the datasets (Dataset Small 1) to optimize all hyperparameters[6] of the model that were not fixed in the experiment. For all subsequent experiments, we then fixed these hyperparameters to the results of the hyperparameter optimization.

In order to give a quantitative answer to RQ 1, we have to compute the following metrics:

Fraction of valid edges: Since, of course, we want that the generated graphs are valid according to the given meta-model, we have to ensure that the generated references are correct. That is, we have to ensure that for a given edge and its source and target node, a corresponding reference will be in the meta-model.[7] We then compute the fraction of valid edges as a metric:

$$\text{validity} = \frac{\text{Number of valid edges}}{\text{Number of edges}}.$$

Total Variation Distance of the Graph Invariant distributions: A graph invariant is a property of a graph that is invariant under isomorphism (cf. Section 3.3.2). Therefore, a good generative model should generate graphs that have similar distributions of graph invariants as the original dataset. Unfortunately, there is no known polynomial-time computable finite set of graph invariants that can capture all properties of a graph (see Theorem 3.3.4). Therefore, we have to choose a set of graph invariants that we think are important for the task at hand. For this experiment, we choose the following graph invariants: Distribution of node labels, edge labels, graph size, number of incoming (in-degree), and number of outgoing edges (out-degree). To compare distributions, we use the Total Variation Distance [203]:

Definition 7.3.1 Total Variation Distance [203].

$$\delta(P, Q) = \frac{1}{2} \sum |P(x) - Q(x)|$$

6 The optimized hyperparameters are learning rate, batch size, KL beta, encoder size, latent size, decoder size, number of layers, dropout, and regularization.

7 We will ignore constraints in the meta-model that go beyond this type correctness, such as quantifiers, or additional constraints given in some constraint language.

The Total Variation Distance ranges from 0 to 1. The lower the value, the more similar the distributions are.

We also compare the Graph Variational Autoencoder approach with "random baselines":

RandomGraphBaseline: Given the size of a graph n, we randomly choose n node labels. For each pair of nodes, we randomly decide if there should be an edge or not.

ValidGraphBaseline: Similar to RandomGraphBaseline, but edges will only be added, if they are valid according to the meta-model.

Experiment 2: For this experiment, we train the Graph Variational Autoencoders first. We then train a binary classifier on the latent space of the Graph Variational Autoencoder. The classifier predicts if a certain input simple change graph contains a pattern subgraph (i.e., our edit operation) or not. The classifier's hyperparameters will be optimized via a hyperparameter optimization on one of the generated datasets that we do not use during the evaluation.

Furthermore, instead of a binary classification, we also want to find out, if the latent representation encodes information about the frequency of a certain pattern subgraph (i.e., the edit operation). We therefore train a simple classifier neural network on the latent space. That is, we fix the maximum number of pattern occurrences to a constant value (i.e., 5). We then use a multi-label classifier to predict the frequency. For this experiment, we only use the best performing layer architecture (i.e., TransformerConv [300] (TRF)) from Experiment 2.

In order to give a quantitative answer to RQ 2, we use the following metrics to evaluate the experiment outcome:

Accuracy: Accuracy is the fraction of correct predictions among all test predictions [362].

$$\text{Accuracy} = \frac{\text{number of correct predictions}}{\text{total number of test predictions}} \tag{7.6}$$

Accuracy can be misleading, for example, in the case of an imbalanced test set. For example in a binary classification task, if 90% of the samples are of class A and 10% of class B, a classifier that predicts all samples as class A will achieve an accuracy of 90%. As common for (binary) classification tasks we therefore also define additional metrics:

TP: True positives (TP) for a given class, is the count of correct predictions for that particular class [362].

FP: False positives (FP) for a given class, is the count of incorrect predictions of that particular class [362].

TN: True negatives (TN) for a given class, is the count of correct predictions for all other classes [362].

FN: False negatives (FN) for a given class, is the count of incorrect predictions for all other classes [362].

Precision: Precision is the fraction of true positives among all positive predictions.

$$\text{Precision} = \frac{TP}{(TP + FP)} \tag{7.7}$$

Recall: Recall is the fraction of correct predictions for a class and all instances of that class.

$$\text{Recall} = \frac{TP}{(TP + FN)} \tag{7.8}$$

F1-score: The F1-score is the harmonic mean of precision and recall. It therefore balances both metrics and combines them in a scalar value. It ranges from 0 to 1, where higher values indicate better classification performance. [362]

$$F_1 = 2 \cdot \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{7.9}$$

For the non-binary classification we calculate the average of the metrics over all classes.

Furthermore, we report the Matthews Correlation Coefficient:

Matthews Correlation Coefficient (MCC): The Matthews Correlation Coefficient (MCC) [116] as a metric that describes the correlation of the prediction with the ground truth.

$$\text{MCC} = \frac{(TP \times TN - FP \times FN)}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{7.10}$$

The advantage of the MCC is that it can easily be extended to multi-class classification tasks and is not biased by imbalanced datasets.

Large Language Models

To answer RQ 3, RQ 4, and RQ 5, we will conduct three experiments.

For every simulated repository in the dataset from Section 7.3.2, we applied the steps of the training phase, described in Section 7.2. We controlled for the number of epochs (i.e., 4 and 6) and the base language model used for the fine-tuning (i.e., text-ada-001, text-curie-001, and text-davinci-003 from the GPT-3 family of language models). Since fine-tuning the text-davinci-003 model is quite expensive (i.e., 0.03US$

per thousand tokens at the time of conducting the experiment), we fine-tuned this model only for the model repositories where perturbation probability equals 100% (the ones which are typically the harder ones). We therefore have to report separately if we also include the text-davinci-003 model in the analysis. We split the datasets (for each of the repositories from Section 7.3.2) in a training and a test set (90% of the samples in the train set and 10% in the test set), so that we can report on the performance of the (textual) completion task the models were trained on.

The total cost for the training via the OpenAI API was 347US$.

Experiment 3: To answer the RQ 3, we use the fine-tuned language models and apply the edit operation candidate generation from Section 7.2. During the edge extension in the generation phase, we checked whether the completion actually corresponds to a correct graph serialization. We also checked whether the edge extension corresponds to a correct extension according to the meta-model, that is, whether the generated graph actually corresponds to a simple change graph. We then report how many of the extensions during the generation were incorrect and how the number of incorrectly generated serializations depends on the base language model, the number of training epochs, the perturbation parameter, and the number of training tokens in the dataset.

Experiment 4: We fine-tune the language models based on a model completion task. Therefore, the procedure from Section 7.2 for deriving fine-tuning data from a model repository together with the training test split described above yield data that can be used to evaluate whether language models can be used for model completion, answering the RQ 4. We investigate the simple change graph completion from two perspectives: First, we investigate the average token accuracy on the test set during the fine-tuning of the language model. The *average token accuracy* gives us the relative number of correctly retrieved tokens. The metric is not aware of any specifics of the dataset. For example, even a single wrong token in a serialization can produce a syntactically wrong serialization while the token accuracy for this can still be high. We therefore also analyze the completion operation candidates from a graph matching perspective. Since generating all completion candidates for all test samples of all fined-tuned language models will be quite expensive, we select two fine-tuned language models and perform the analysis for them. From the set of generated completion operation candidates, we especially look at two candidates: the one providing "the best completion" and the "top-ranked completion" using the edge-scaled ranking metric. In some sense, these two selected candidates give us an upper and a lower bound for the performance on the software model completion task. To make completions comparable, we assign a numeric value to them. This is possible, since we define completion candidates isomorphic to the correct graph to be better than completion candidates that are too large (i.e., the ground truth is a subgraph of the completion candidate), which themselves are better than completion candidates

that are too small, which again are better than incorrect completion candidates (i.e., some edges missing and some additional edges).

Experiment 5: To answer RQ 5, we evaluate the generation of edit operation candidates from the fine-tuned language models. To this end, we apply the approach from Section 7.2 to the synthetic model repositories. Since we know the edit operations that have been applied, we can directly look for the applied edit operations in the list of edit operation recommendations. We count the number of correct edit operation candidates that have been generated. There are three correct edit operations in total, and each of them has been applied in every software model to our synthetic dataset (cf. Section 7.3.2). Additionally, we investigate how the number of correctly retrieved edit operations depends on repository as well as language model parameters. We will also investigate the costs for the generation of edit operation candidates.

We then compute the different ranking metrics (i.e., language model probability, the probability scaled by the factorial of the number of edges, probability scaled by the number of edges, and compression-like metric) for all generated edit operation candidates and compare the different rankings based on the given ground truth, that is, the rank of the known applied edit operation. To compare the different ranking metrics, similar to our evaluation of OCKHAM in Chapter 6, we use the *mean average precision at k* (MAP@k) given in Definition 6.3.1 (repeated here for readability):

$$\text{MAP@k} := \frac{1}{|D|} \sum_{\text{D}} \text{AP@k} \ ,$$

where D is the family of all datasets (one dataset represents one repository) and AP@k is defined by

$$\text{AP@k} := \frac{\sum_{i=1}^{k} \text{P}(i) \cdot \text{rel}(i)}{|\text{all correct simple change graphs}|} \ ,$$

where $\text{P}(i)$ is the precision at i, and $\text{rel}(i)$ indicates if the candidate at rank i is relevant.

7.3.4 Results

As before, we will first present the results of the experiments for the graph neural networks and then for the large language models.

Graph Neural Networks

We will present the results for Experiment 1 and Experiment 2 individually.

Experiment 1: In Experiment 1 we were interested in two things: The validity of generated simple change graphs according to the meta-model (i.e., the type correctness), and the similarity of the generated and the original simple change graphs.

Type correctness: Table 7.2 depicts the average type correctness of the generated graphs and the standard deviation across the 8 datasets (cf. Section 7.3.2).[8] We performed a right-tailed t-test (with significance level $\alpha = 0.01$) of the mean validity of the Graph Variational Autoencoder approaches against the RandomGraph baseline. We found all approaches to significantly outperform the random baseline.

Table 7.2: Type correctness of the generated graphs, the meta-model validity without failed runs are added in brackets

Method	Validity (mean)	Validity (standard deviation)
TransformerConv	0.9982*	0.0025
GINEConv	0.9971*	0.0038
GATConv	0.9932*	0.0172
RandomGraph Baseline	0.0468	0.0021

* $p < .01$

Furthermore, we took a closer manual look into the generated simple change graphs that were not correct according to the meta-model. This analysis did not reveal any clear pattern.

Distributions: Table 7.3 depicts the results of the comparison of the distributions for the five selected graph invariants. We computed the mean values of the total variation distance across all 8 datasets and random seeds. We also performed left-tailed t-tests for each of the graph invariant against the baseline approaches (significance level $\alpha = 0.01$). For all graph invariants and all approaches, we found that the total variation distance is significantly lower.

Table 7.3: Mean values of total variation distance for each of the selected graph invariants. Lowest (i.e., best) values are marked in bold. The last row shows the average across all approaches without baselines.

Method/Invariant	Edge labels	Node labels	Graph size	In-degree	Out-degree
TransformerConv	**0.028**	**0.036**	**0.190**	**0.103**	0.052
GINEConv	0.054	0.073	0.433	0.168	0.099
GATConv	0.041	0.040	0.226	0.130	**0.048**
RandomGraph Baseline	0.401	0.213	0.560	0.787	0.648
ValidGraph Baseline	0.482	0.208	0.533	0.282	0.167
mean of TRF, GIN, GAT	0.102	0.095	0.363	0.169	0.119

8 Note that we also rerun every experiment with different random seeds. The mean and standard deviation are then computed over all datasets and seeds.

Experiment 2: For Experiment 2, we have to distinguish between the binary classification results and the multi-label results (i.e., predicting the edit operation frequency).

Binary Classification Probing: Table 7.4 depicts the results of the binary classification task.

Table 7.4: Several performance metrics for the pattern probing binary classification task. The best values are marked in bold.

Method	Accuracy	Precision	Recall	F1	MCC
TransformerConv	**0.857**	**0.862**	**0.865**	**0.855**	**0.726**
GATConv	0.769	0.774	0.766	0.761	0.540
GINEConv	0.776	0.804	0.776	0.765	0.576

One alternative explanation for a good performance in the classification task is that the classifier learned to memorize if a certain latent representation is associated with a certain pattern. To rule out this possibility, we assigned new (incorrect) labels on the dataset (only based on the size of the simple change graph and the size of the pattern subgraph). We then trained the classifier on these new labels. The results are given in Table 7.5.

Table 7.5: Several performance metrics for the pattern probing binary classification task with random labels. This serves as a baseline to measure memorization classification in the latent space.

Method	Accuracy	Precision	Recall	F1	MCC
TransformerConv	0.550	0.566	0.550	0.520	0.115
GATConv	0.584	0.599	0.584	0.563	0.182
GINEConv	0.595	0.606	0.595	0.576	0.200

Multi-label Classification Probing: We perform the multi-label classification to predict the frequency of the pattern subgraph (i.e., the edit operation) only for the best performing Graph Variational Autoencoder—using the TransformerConv (TRF) layer architecture. The results of this multi-label classification are given in Table 7.6.

Large Language Models

We will next present the results for the experiments relating to the large language models.

Table 7.6: Performance metrics for the pattern probing multi-label classification task (i.e., to predict the frequency of a pattern subgraph). Furthermore, the classifier trained on the incorrect labels is shown as GraphSize Baseline.

Method	Accuracy	Precision	Recall	F1	MCC
TransformerConv	0.696	0.369	0.394	0.370	0.463
GraphSize Baseline	0.193	0.105	0.163	0.120	0.004

Experiment 3: For Experiment 3, we analyze if type-correct candidates can be generated from the large language models using our approach from Section 7.2. For 51.8% of the simulated repositories, the generation procedure produced exclusively valid graphs, for 48.2% it solely produced type-correct simple change graph (i.e., also correct with respect to the meta-model). On average, 2.26 invalid graphs are generated in the generation phase, with a minimum of 0 and a maximum of 30. These invalid graphs are then discarded in the further generation phase. A constraint in the given EdgeList serialization (see Section D.2) is that a node with a given id has to appear always with the same label (i.e., the node labels are redundantly encoded in EdgeList). The only type of violation against a valid EdgeList encoding we have encountered in this experiment was that this correspondence of node id and node label has been violated. A manual inspection of the data for the model repositories with a large amount of invalid generated graphs (> 5) reveals that these are the "smaller" datasets with mostly a high perturbation. For example, the repository with only 10 revisions, 11 applied edit operations, and a perturbation of 100% is the one with the maximum of 30 invalid graphs.

In Table 7.7 we report the correlation coefficients (Spearman[9]) between invalid graphs and invalid simple change graphs and the base language model (BM), the number of training epochs, the perturbation probability (P), and the number of training tokens of the dataset (T). For the correlation with the base language model, we sort the base models according to their size (i.e., text-ada-001: 0, text-curie-001: 1, text-davinci-003: 2).

We observe significant positive Spearman correlation between the number of invalid generated graph serializations and the perturbation parameter. Furthermore, there is a significant negative Spearman correlation between the number of invalid generated graph serializations and the number of tokens in the training set.

Experiment 4: Before we analyze if and to what extent the ground truth edit operation will be generated from the large language models, we first consider the contextualized generation (i.e., a kind of completion task).

At the token level, we find an average token accuracy of 96.9%, with a minimum of 92.1%, and a maximum of 99.0% on our test data set.

9 We use Spearman correlation, since we can not assume that there are linear dependencies.

Table 7.7: Correlation (Spearman) between invalid graphs generation with the parameters base language model (BM), the number of training epochs (Epochs), the perturbation probability (P), and the number of training tokens of the dataset (T).

	BM	Epochs	P	T
#Invalid	−0.08	−0.10	0.35 **	−0.33 *
#Invalid meta-model	−0.22	−0.05	0.31*	−0.22
#Invalid (+ text-davinci-003)	−0.28	−0.11	–	−0.56 **
#Invalid meta-model (+ text-davinci-003)	−0.24	−0.13	–	−0.56 **

$^{*}\ p < .01$ $^{**}\ p < .001$

Table 7.8: Correlation coefficients (Spearman) of the completion candidate score with the number of omitted edges in the sample, the number of total edges of the correct simple change graph, and the number of completion candidates that have been generated.

	#Omitted Edges	#Total Edges	#Completions
Score (best rank)	−0.69 *	−0.38 *	−0.75 *
Score (best candidate)	−0.29 *	−0.33 *	−0.21 *

$^{*}\ p < .001$

Only 2.71 completion candidates are generated, on average. For a large number of samples, only one completion operation candidate has been generated. In all of theses cases, the only candidate has also been the correct one. On average, in 87.60% of the samples, the correct completion is among the candidates, with an average rank of 1.55 (w.r.t. the edge-scaled ranking).

In Table 7.8 we report on the Spearman correlation coefficients of the score of the completion candidates with the number of omitted edges (i.e., the ones that have to be computed), the total number of edges of the ground truth simple change graph, and the number of completions that have been generated.

We see significant negative correlations between the score (of the best generated candidate and the best ranked candidate) and the number of edges that have to be completed, the number of edges from the full original simple change graph, and the number of completions candidates that have been generated.

Experiment 5: In Experiment 5, we analyze if the ground truth edit operation can be retrieved via the approach from Section 7.2.

We find that, on average, out of the three applied edit operations, we could retrieve 2.17 for the text-ada-001 model, 2.13 for the text-curie-001 model, and 1.00 for the text-

Table 7.9: Correlations between the correctly retrieved edit operations and repository as well as language model parameters.

	BM	Epochs	P	E
#Correct	−0.33 *	−0.03	−0.80 *	0.10
#Correct (+ davinci)	−0.22	0.08	–	0.28

* $p < .001$

davinci-003 model. The text-davinci-003 model has only been trained to the datasets that appeared to be difficult for text-ada-001 and text-curie-001 (i.e., perturbation probability of 100%).

In Table 7.9, we list the Spearman correlations of the correctly retrieved edit operations with the base model (BM), the number of training epochs, the perturbation probability (P), and the number of applied edit operations between two model revisions (E).

Regarding the different ranking techniques, we list the MAP scores for the 4 different ranking metrics in Table 7.10. Furthermore, we compare the average precisions obtained through the different ranking. We can observe a high significant ($p < .001$) Spearman correlations coefficients > 0.65 among all of them, the largest one between compression metric and node factorial scaled probability metric (0.90).

Table 7.10: MAP for the different evaluated ranking metrics. Grey background indicates the best MAP among all metrics.

	Compression	Factorial	Probability	Edges-Scaled
MAP@3	0.32	0.20	0.13	0.33
MAP@5	0.38	0.29	0.17	0.36
MAP@10	0.41	0.32	0.24	0.40
MAP@∞	0.42	0.33	0.25	0.41

The average generation cost for text-ada-001 was 0.45 Cent, for text-curie-001 3.68 Cent, and for text-davinci-003 51.84 Cent.

7.3.5 Discussion

We will now discuss our observations. We will first address the research questions formulated in Section 7.3.1. Then, after a discussion of threats to validity, we will put the results in the context of the research goal and the perspective of this thesis.

Graph Neural Networks

In RQ 1 we were interested in how well the Graph Variational Autoencoder approach can resemble the original simple change graphs. In our experiment we investigated type-correctness of the generated graphs, and we compared five graph invariants (edge label distribution, node label distribution, graph size, in-degree distribution, and out-degree distribution). We also experimented with 3 different layer architectures. Except for the out-degree distribution, the TransformerConv (TRF) architecture outperformed the other two architectures. From the experiments we can also clearly see that generated graphs are mostly correct (i.e., also type-correct) simple change graphs. Regarding our selected graph invariants, all approaches clearly outperform the baselines considered. In general, the variation distances for all five invariants[10] are rather low, even though, especially for the graph size, there is a clear difference between the generated graphs and the original simple change graphs.

RQ 1: *How well can the proposed Graph Variational Autoencoder approach resemble the original simple change graph distribution?*
The proposed approach is able to generate mostly correct simple change graphs. The variation distances for the selected graph invariants are rather low. Nevertheless, we can not claim that the approach is able to generate simple change graphs that resemble the original simple change graphs.

In RQ 2 we were interested in whether the proposed Graph Variational Autoencoder approach "memorizes" patterns that exist in the training data. In our experiments we investigated *probing* classifiers. That is, we trained classifiers on the latent space of the Graph Variational Autoencoder to predict if a certain graph contains a pattern and how frequent the pattern is. From the experiments we can see that the probing classifiers are able to predict the presence of the patterns to a certain extent, the frequency of the patterns, however, is not predicted well (given a rather low F1 score). As for the RQ 1, the TransformerConv architecture outperformed the GINEConv, and GATConv convolutional layers.

RQ 2: *Does the proposed Graph Variational Autoencoder approach "memorize" edit operation that have been used to construct the training data?*
Our experiments provide some evidence for the hypothesis that the proposed Graph Variational Autoencoder approach is able to memorize patterns that exist in the training data. The probing classifiers are able to predict the presence of the patterns to a certain extent. The frequency of the patterns, however, is not predicted well.

10 It is important here to emphasize that the observations have been made only for the five graph invariants reported here. Indeed, as it turned out later, the results are not generalizable to other graph invariants. We will discuss this point later in detail.

Large Language Models

Regarding the RQ 3, from our observations, we can conclude that—given the approach from Section 7.2—we generate mostly valid graph serializations. Indeed, even for a majority of the simulated repositories, we do not generate invalid graph serializations at all, and, on average, 2.26 invalid candidates per generation phase. The analysis also shows that the repositories for which we also get invalid graph serializations are the smaller ones (in the number of training tokens), with a high perturbation. The dependency on the size of the repository and the perturbation is also significant (as can be seen from our results in Table 7.7). This suggests, that one should use larger repositories or even pre-train the language model with simple change graph serializations from other repositories. We also observe a small negative but insignificant correlation w.r.t. the size of the base language model that has been used for the fine-tuning. This suggests that larger base language models might perform better in the generation of simple change graph. Given that text-davinci-003 is 50 times as expensive as text-ada-001, and this correlation is not significant, we can conclude that text-ada-001 is an acceptable choice for the base language model for the generation of simple change graphs.

RQ 3: *Using our approach, are language models capable of generating correct simple change graph graph serializations?*
Overall, invalid graphs are generated only rarely. Smaller repositories are more likely to lead to invalid graphs than larger repositories.

In RQ 4, we were interested in how well language models can provide correct (i.e., correct format and type correct) auto-completions for software models. From the results, we can clearly conclude that—for a majority of the samples—the correct completions have been generated. Furthermore, the number of generated completion operation candidates is typically low and the correct completion operation (if among the candidates) is typically top ranked. The larger the simple change graph and the more edges we omit for the completion, the worse the score of the completion candidates. Anyway, for the larger graphs, a subgraph of the correct completion was among the candidates in most cases.

RQ 4: *Using our approach, are language models capable of providing correct auto-completions for software models?*
A manageable amount of model completion candidates are generated. In a majority of the cases the correct completion has been generated by the language model. Larger simple change graphs (in number of edges) and a larger number of omitted edges are more challenging than smaller ones, which is rather intuitive.

RQ 5 concerns the reconstruction of edit operations from the large language model. In our experiments, using the approach from Section 7.2, we were able to retrieve two and, in some cases, even all applied edit operations. The perturbation seems to be the major influencing factor, increasing the difficulty significantly. Using larger language models (i.e., text-davinci-003) did not improve the results. We observed even a decrease of the number of retrieved edit operations with increasing language model size. The costs for using the language models are definitely acceptable (0.45 Cent to 51.84 Cent), especially given that the more expensive language models do not perform any better.

From the results of Experiment 5, we can furthermore see that, for $k > 3$ (where k is the number of considered generation candidates), the compression metric outperforms the other metrics. Since the compression metric is expensive to calculate, we also tried to recompute it from the probability given by the language model. Although we can observe high correlations among the average precision for different rankings, none of the three metrics yields the same ranking as the compression-based ranking. For practical purposes, the edges-scaled probability metric is most feasible, since it gives results close to the compression metric and does not require any expensive calculations.

RQ 5: *Using our approach, can edit operations be reconstructed from the language model?*
We were able to retrieve edit operations from a fine-tuned language model. However, with increasing perturbation probability, the approach yields worse results. We can confirm our earlier findings (cf. Chapter 6) that the compression metric seems to be a good metric to select relevant edit operations.

Threats To Validity

General Threats: We will first discuss the threats that hold for all experiments, and then discuss more specific threats.

One general threat to validity is that there is no direct method, to understand the information encoded in a neural network.[11] We therefore have to rely on indirect methods, such as probing or constructive approaches such as our language model pattern generation approach from Section 7.2. Of course, these indirect method must always be taken with care, because even for positive results, confounding factors could be an alternative explanation (for example memoization in the latent space). We also only limited our experiments to a concrete choice of methods, other methods could have led to other results.

11 It is a bit ironic, because a similar situation holds for human implicit pattern knowledge. We argued earlier that many edit operation are probably rather implicit then explicit knowledge.

Furthermore, the experiments conducted in this chapter were only performed on the SYNTHETIC dataset and are therefore not necessarily generalizable. Especially, is stays unclear, if the results generalize to settings with more complex meta-models or many more existing edit operations. The SYNTHETIC dataset gives us a controlled experiment setting, this way, increasing internal validity.

Anyway, for the current state of research in the field, the results are promising and show that at least to *some extent* the neural network-based approaches from Section 7.2 seem to capture pattern knowledge. We believe that even with other dataset or other methods, the general direction of the results would be comparable.

Graph Neural Networks: Regarding our experiments with the Graph Variational Autoencoder, we had a large hyperparameter space. We limited the hyperparameter optimization to a single dataset, which may have led to overfitting. The results for other datasets seem stable though. We are therefore confident that our hyperparameter choice is reasonable.

Regarding the choice of the convolutional layer architecture, other layer architectures could have led to other results. We chose the most promising architectures from the literature promising to capture global structural properties of graphs. All three selected architectures show similar results, and we are therefore confident that the results are not due to the specific choice of the architecture.

Furthermore, the Graph Variational Autoencoder Architecture we have chosen has a fixed size input. It is not clear, how the architecture would perform on larger or more diverse graphs.

Large Language Models: We have evaluated the edit operation candidate generation approach in a controlled experiment setting. However, further experimentation and evaluation is required before it can be considered for implementation in real-world software engineering projects. One of the main reasons to take such a staged approach is that the application of language models to large-scale industrial experiments requires training on large model repositories. Training on all possible simple change graphs and serializations is infeasible from a cost perspective.[12] We therefore have to better understand large language models in the domain of model generation to use a suitable sampling in real-world applications. Here, we made the explicit decision to trade-off external validity for internal validity [302] before moving on to the next stage, that is, an application to real-world models. The purpose of the present study is to gain some preliminary knowledge about whether language models can learn patterns from the data and whether it is possible to extract the patterns from the language models.

With respect to internal validity, we have chosen those properties and parameters that intuitively have the highest impact on fine-tuning language models, although

12 In Chapter 8, we will take a different approach to this problem by using a generative model without expensive fine-tuning.

we cannot control for any arbitrary property of the modeling language or the model repository (because of combinatorial explosion). Several design decisions (e.g., which parameters to fix and which to vary) have to be made before language models can be applied to the domain of software model completion (e.g., serialization strategy, graph encoding, choice of the base model, etc.). Other design decisions could have led to other conclusions and there is still room for improvement and optimization of language models for edit operation mining and model completion. Furthermore, the base models (i.e., GPT-3) that were fine-tuned in this work are only available through an API. We wanted to conduct this study with state-of-the-art models, and at the time the experiments were conducted, GPT-3 was the state-of-the-art without significant competition. However, it would also be interesting to compare a larger set of language models on the above tasks, although this is well beyond the scope of the present study.

General Discussion

First, we want to put the results from this chapter (regarding edit operation candidate generation) into the context of the results from Chapter 6. As we have explained in the introduction, the approach from Chapter 6 is not a promising direction for context-aware model auto-completion. Regarding edit operation mining, frequent subgraph mining is challenging from a computational or memory complexity point of view. This sometimes requires omitting large simple change graphs in the mining and extending the approach to co-occuring changes not feasible. The reason for this is that subgraph matching requires to try many possible combinations that vertices from the subgraph can be matched to the super graph. Using language models, we circumvent this issue and train on sequential—instead of graph-like—data. The training resources (and therefore also cost) scale linearly with the number of tokens. A graph would be "equivalent" to all of its possible serializations (i.e., approximately $|E|!$). Training a language model on all these serializations would then be infeasible. Anyway, using a language model as described in Section 7.2, we get control over the number of serializations, and we have seen that considering only one serialization per graph yields promising results. Also, in our experiments from this chapter, we can reconfirm the results from Chapter 6 that the compression metric seems to be a good metric to select relevant edit operations. It also has to be said that neural networks can be seen as a method for (lossy) compression of training data. Even though not formally equivalent, it is fair to say that OCKHAM and generative neural network approaches follow a similar strategy.

Another observation that we made during our experiments is that the Graph Variational Autoencoder approach is rather unmature. For example, it is unclear how the approach scales to larger input graphs. Furthermore, the loss function explicitly encodes several graph properties that the reconstruction loss focuses on. In another later pilot experiment, we therefore explicitly looked into graph invariants that are not reflected by the reconstruction loss of the implementation, and, indeed, some

of these graph invariants, such as the clustering coefficient, were poorly replicated in the generated graphs. It would therefore be more promising, to explore some autoregressive training for graph neural networks, similar to the generative pre-trained transformer architecture that we used for large language models. This way, we would not have to encode all important properties (which we typically do not even know) into the loss function. At the time of experimentation, we were not aware of any working autoregressive generative graph neural network architecture that could be used for our purpose. Anyway, for this chapter, we were rather interested in whether generative neural networks do capture pattern knowledge. The results from our experiments suggest that indeed, the Graph Variational Autoencoder approach is able to memorize patterns to a certain extent, that exist in the training data.

Regarding the large language models, unfortunately, because we limited ourselves to use state-of-the-art models which were only available via an API, we could not look deeply inside the models. This might be interesting future work to explore a similar probing approach to large language models.

Regarding the first goal of this chapter, we can conclude that large language models and graph neural networks seem to encode pattern knowledge. For the second goal—the generation of edit operation candidates—our large language model approach performs promisingly on the SYNTHETIC dataset. More experiments on other datasets are necessary to evaluate the generalizability of our results.

7.3.6 Related Work

One area of research related to inference of edit operations is automatic "by-example" program synthesis [120] such as Flash Meta [261]. From the perspective of this kind of research, in the approach presented in this work, we try to learn a grammar. We haven't evaluated language models yet for identifying functional relationships between attributes in software models. Instead, our approach is able to detect patterns in the data and does not need any example pairs to derive these patterns. Therefore, the program synthesis approaches are more similar to the classical model transformation by-example approaches. For the future, a combination of symbolic inductive programming approaches with neural approaches as the one presented here might be an interesting field of research.

In the area of molecular generation, recently several approaches have been proposed that directly work on graphs instead of textual representations [77, 196, 304]. Kwon et al. [196] and Simonovsky et. al [304] also propose a Graph Variational Autoencoder for molecular generation, similar to our approach. Indeed, the work by Kwon et al. [196] has been the implementation basis for our work. With MolGAN [77], De Cao et al. propose an alternative approach based on generative adversarial networks (GANs) for molecular generation. Also in the molecular generation domain, many approaches work on linearized representations of the molecular graphs (e.g., SMILES representation) [195], instead of directly using graph neural networks. In

cheminformatics and bioinformatics, language models have been investigated for molecular generation [25, 364].

Another related area is Knowledge Graph extraction. Here, researchers investigate the extraction of knowledge graphs from language models [108, 316]. Our idea of extracting edit operations from a fine-tuned language model is similar in that we also try to extract "learned knowledge" from a language model. Anyway, our application domain is a different one.

Regarding the application of natural language processing—and language models, in particular—to software models, there have been some research activities in the Model-driven Engineering community: Burgueño et al. propose an NLP-based architecture for the auto-completion of partial domain models. They do not employ language models in their approach and instead use other natural language processing approaches (i.e., word embedding similarity) to recommend words, which are then transformed into model elements in a post-processing [48]. Weyssow et al. use a long-short-term memory neural network architecture to recommend metamodel concepts but they do not generate entire model completions [345]. Tsigkanos et al. propose using language models to extract variables from user manuals for metamorphic testing [67].

Furthermore, it needs to be mentioned that for source code, the use of language models for code completion is outperforming other approaches and capable of generating code from natural language input [62, 68]. Sobania et al. [305] compare large language model-based GitHub CoPilot to genetic programming based program synthesis. Wan et al. [337] study if abstract syntax tree representations can be retrieved from language models trained on code.

7.4 Conclusion

For this chapter, our goal was to understand to what extent pattern knowledge is present in a generative neural network model, which—according to the theory—would be a good technology to support software model evolution. We also wanted to have some preliminary results on the feasibility of reading out the pattern knowledge from the model.

We conducted five experiments—two for graph neural networks and three for large language models—to pursue this goal. In our experiments, we found initial evidence that pattern knowledge is present in the generative neural network models parameters. For graph neural networks, we observed a good classification performance of a probing classifier that, given a latent representation, predicts if a certain edit operation was applied or not. Regarding large language models, we found that the models were particular good at completing simple change graphs, which were generated from a synthetic dataset. We were also able to generate edit operations

that were applied to generate the synthetic dataset, although not always all of them (i.e., not for all repositories).

To increase internal validity of the results, the experiments have been conducted for the SYNTHETIC dataset, which can be considered to be a rather "simple" dataset. This kind of research, that is, to identify and extract pattern knowledge—and edit operations, in particular—in generative neural networks, is still in a very early phase.

For the Graph Variational Autoencoder, at the current state of technology, we see several limitations: the fixed input size, a large hyperparameter space, numerical instability, and the need to encode *important* graph invariant into the loss function. Especially the last point is a challenge, because it is not clear how to encode the information that is important for certain domain-specific edit operations into the loss function. For our goal of tackling the long tail of domain knowledge, we therefore do not see the Graph Variational Autoencoder as a suitable technology at the current state of technology. Although the comparison to large language models is not fair—because these are huge neural networks compared to the Graph Variational Autoencoder we have chosen—we see that large language models are more promising to pursue our goal. A combination of both technologies might be a promising approach for future research, combining the strengths of both technologies (e.g., GNNs for reference recommendations and LLMs for tasks like class or attribute recommendation).

Note that the generative neural network models we have used in this chapter follow a similar "compression principle" as the minimum description length principle (cf. Section 3.2.2): Starting with an empty model a generative neural network model can generate the next revision. In the latent space, we would then typically make some error due to compression. A model could then be described by the generative process in the compressed latent space plus these errors made in the latent space. This yields a new training task optimization formulation for generative neural network models, which could be interesting for future research.

Since we made promising observations in the experiments described in this chapter, for the rest of this thesis, we will focus on large language models and investigate their application to providing context-aware model evolution support.

8

Towards Handling the Long Tail of Domain Knowledge

No man ever steps in the same river twice, for it's not the same river and he's not the same man.

— Heraclitus

This chapter shares material with the International Conference of Software Engineering (ICSE) 2025 paper titled "Software Model Evolution with Large Language Models: Experiments on Simulated, Public, and Industrial Datasets" [326].

Modeling structure and behavior of software systems plays a crucial role in the industrial practice of software engineering. As with other software engineering artifacts, software models are subject to evolution. Having intelligent modeling assistants (see Section 2.5) that support modelers in evolving software models—similar to GitHub Copilot in the source code domain—is still an open challenge, though. This chapter will explore the potential of large language models for model (auto-)completion, as a particular use case that falls into the responsibilities of an intelligent modeling assistant. In particular, we propose a concrete approach, RAMC, that applies large language models with retrieval-augmented generation to software model histories for this purpose. The chapter will present experiments on all datasets from Chapter 5—INDUSTRY, REPAIRVISION, and SYNTHETIC—to understand the merits of the proposed approach. We find that large language models are indeed a promising technology for supporting software model evolution (62.30% semantically correct completions on real-world industrial data and up to 86.19% type-correct completions). The general inference capabilities of large language models are particularly useful when dealing with concepts for which there are few, noisy, or no examples at all.

The positive results in this chapter can also be seen as evidence for the more general hypothesis that parametric generative models are suitable for use cases like model auto-completion (cf. Hypothesis 2).

This chapter presents how large language models can be used to provide contextualized *auto-completion* for software models based on the theory from Section 3.4. During the review of this thesis, it became clear that speaking of *auto-completion* understates the importance of this use case. Probably, this is one of the reason for the emergence of the term *copilot* in similar applications (e.g., the source code domain or the Microsoft Copilot in Office 360). Due to its underlying prompt engineering, other use cases such as explaining or fixing a diagram can be implemented mainly by adjustments of the corresponding prompt. Anyway, in this chapter, we will strive for focus and stay in the auto-completion setting.

In a summary, with this chapter, we make the following contributions:

- We propose a retrieval-augmented generation approach, RAMC, for software model completion.
- We evaluate RAMC qualitatively and quantitatively on all three datasets from Chapter 5, including an industrial application, one public open-source community dataset, and one controlled collection of simulated model repositories, including comparing our approach with the most recent advancements in model completion [58] as well as to the alternative of fine-tuning a pre-trained large language model. For all three datasets, we find that large language models are a promising technology for software model completion, with up to 86.19% correct completions (for the synthetic dataset) and 62.30% of semantically correct completions on the industrial dataset. Notably, our approach improves significantly over the state of the art [58]. Furthermore, it appears that fine-tuning can be an alternative to retrieval-augmented generation that is worthwhile investigating.

8.1 Model Evolution and Model Completion

The goal of this chapter is to study the evolution of software models via large language models, that is, we will have a closer look into software model auto-completion. In this section, we will motivate this goal.

8.1.1 Generative Models for Software Model Evolution

As we have seen previously, from the perspective of the modeling tool, we can understand the evolution of a single software model as a sequence of *edit operations*:

To change or evolve the model, the user executes edit operations (e.g., using mouse clicks and keyboard strokes) provided by the modeling tool. In Example 3.4.1 in Section 3.4, we have seen how this evolution corresponds to a generative model (e.g., a Markov process).

Supporting tool users in accomplishing this evolution of software models is clearly desirable in practice [50, 79, 322]. For the evolution of software models, modeling tools typically provide an initial set of edit operations (e.g., adding an attribute to a model element). Nevertheless, since the usage of a (domain-specific) language is also subject to evolution and since (project-specific) usage patterns might emerge, this initial set of edit operations is likely not exhaustive. For example, in object-oriented design, design patterns [106] are widely used and are not part of UML [247], but could, in principle, be provided as edit operations by a UML modeling tool.

In Chapter 7, we discussed that simple pattern mining is not sufficient: To account for the long tail of specific context, and combinations of edit operations that a user might want to apply, a rather "continuous" generative model is required. We compared this to the situation of language modeling, where n-grams are also not sufficient to cover the long tail of natural language.[1] Clearly, from the perspective of software model evolution, it is desirable to have *context-dependent auto-completions*, rather than utilizing a fixed set of edit operations.

8.1.2 Large Language Models as Generative Models for Software Model Evolution

Large Language Models (cf. Definition 3.4.3) are able to handle very dense, distributed representations of the input, and can adjust their output token probabilities to a given context (e.g., using attention mechanism [330]). In Chapter 7, we have seen that this technology even seems to be able to capture pattern knowledge in the model repositories to some extent.

For source code, modern integrated development environments already support writing and evolving source code by *(auto-)completion*. Most notably, the use of large language models has become state-of-the-art for the auto-completion of source code [10, 11, 62, 96, 338, 352].

The world of software models seems to be lagging behind, and no general approach for software model auto-completion is ready for industrial application. It has been even argued that the so-called cognification of use cases in model-driven software engineering might turn the difference between (perceived) added value and cost from negative to positive [50].

We posit that generative large language models exhibit a deep understanding of language and hold comprehensive knowledge across various domains, which is a

1 Even single words in natural language follow already the so-called *Zipf's law*, which is a long tail distribution.

result of their training on vast corpora. This capability enhances their potential to interpret and complete software models effectively, which usually encompass a vast amount of natural language data.

8.1.3 Model Completion: Definition and Terminology

In Definition 3.5.2, we have defined model completion as a function that, given a modeling step (also called transformation) $m_1 \xrightarrow{\varepsilon} m_2$, returns another (complete) step $m_1 \xrightarrow{\gamma} m_3$. Example 3.5.1 gives a concrete example of a model completion.

If we think of ε as an incomplete step, and γ as a (completed) edit operation, the term "model completion" make sense. Definition 3.5.2 is very general, and we could even think of ε as a complete step. For example, suppose in a system model, we are changing the hardware platform (e.g., as part of an architectural diagram) and the auto-completion might suggest adjusting encryption schemes to those supported by the platform. In this case, γ would be more than just a "completion" and can even represent functional evolution or a context-specific suggestion. Still, for historic reason, we often refer to the task studied in this chapter as *model completion*, or *model auto-completion*, but it has to be said that, in some situations, this is can be an understatement.

8.1.4 Problem Setting

For this section, we will study model completion using large language models with fine-tuning and so-called retrieval-augmented generation in detail.

While recent research suggests that large language models could be utilized for model completion [58], we go beyond and utilize model evolution data from model repositories to capture real-world complexities. It is important to note that, in our work, we explicitly acknowledge the complexity of real-world data, which is due to the close collaboration with our industry partner and the case study from Chapter 4.

8.2 Approach

In this section, we describe RAMC—our approach of *how* to employ large language models to (auto-)complete software models.

8.2.1 Running Example

Consider the motivating example depicted in Figure 8.1, which originates from one of our datasets, REPAIRVISION, further explained in more detail in Chapter 5.

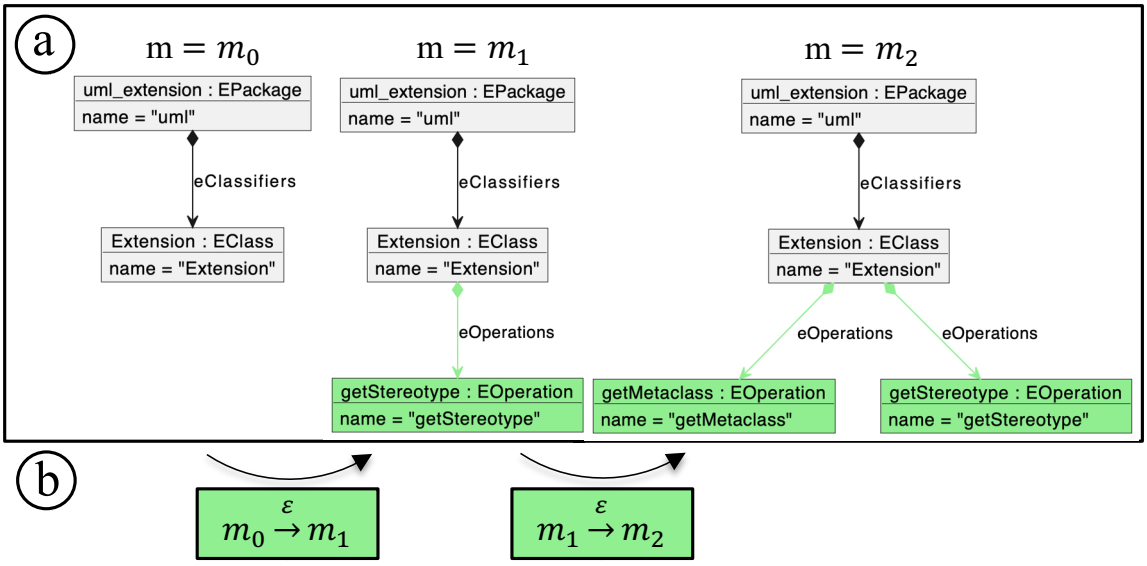

Figure 8.1: Iterative editing scenario in abstract syntax from the real-world REPAIRVISION dataset. Grey color indicated preserved elements, while green color represents created elements.

In ⓐ, we show the evolution of its abstract syntax graph.[2] In this evolution scenario, a modeler adds the UML Profiles mechanism (cf. UML specification [247], Chapter 12.3) to the ECORE meta-model[3] of UML 2.5.1. Step by step the modeler extends the existing UML meta-model with additional functionality, currently focusing on the EClass Extension in the UML package. In a first step, the modeler adds an operation getStereotype (responsible for accessing the Sterotype of the extensions associated with an element in the (meta-)model). As defined in the UML specification [247], every extension has access to the Metaclass it extends, realized in ECORE by the EOperation getMetaclass. This EOperation is implemented by the modeler in a second step. These steps in the evolution of the UML meta-model could be performed via edit operations by a human user, or likewise, recommended in the form of a model completion (as depicted in ⓑ of Figure 8.1).

8.2.2 Overview and Design Choices

Utilizing large language models for software model completion gives rise to several challenges addressed by RAMC: how to provide context, such as domain knowledge,

2 Due to obvious space constraints, only a small part of the original model (only one out of 256 classifiers and 2 out of 741 operations) is shown.

3 UML, according to the Meta-Object Facility [245], is itself a model according to its meta-metamodel, ECORE, and therefore covered by the present work.

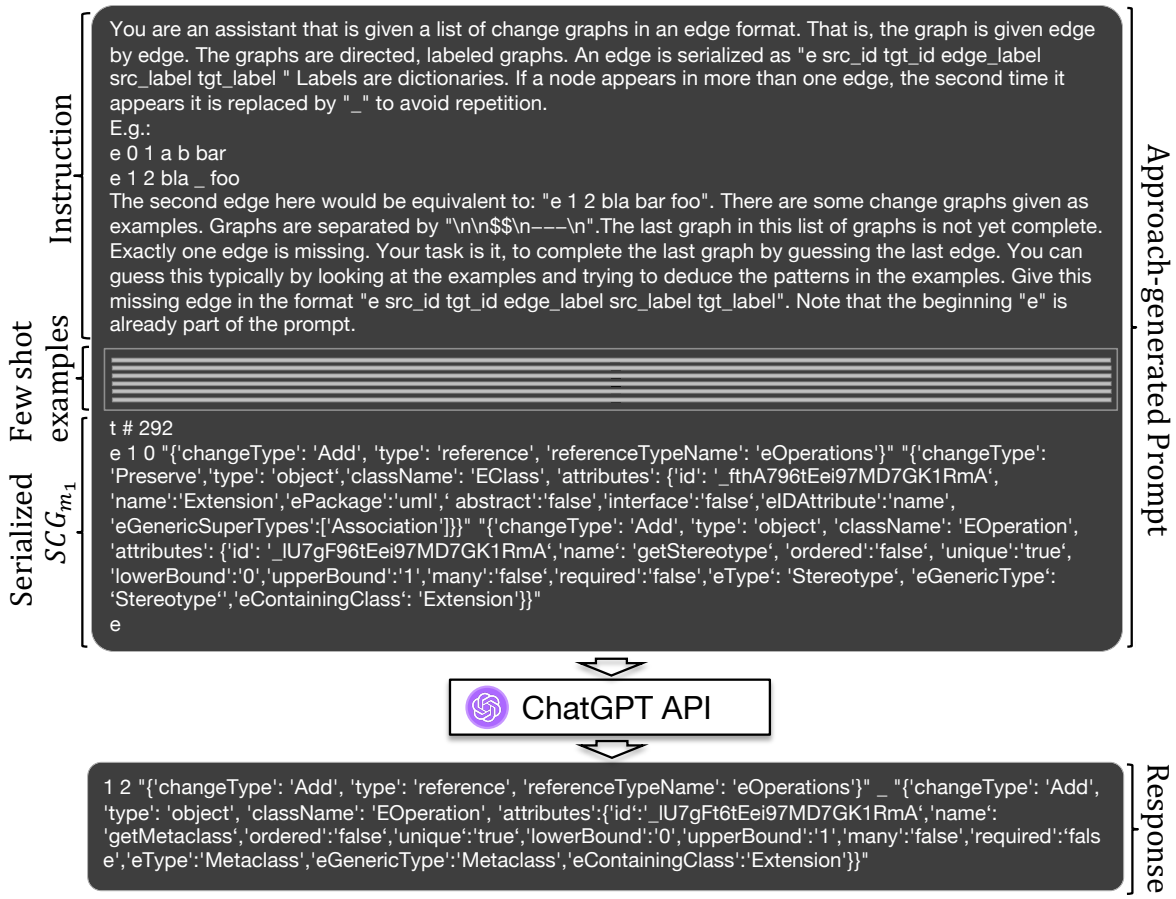

Figure 8.2: Detailed prompt and simple change graph serialization of the RAMC approach corresponding to the example given in Figure 8.1. The full few-shot examples are provided in Section E.3.4 due to their extensive size.

to the large language model, how to serialize software models, and how to deal with limited context[4]?

Regarding context, we opt for retrieval-augmented generation (RAG), and compare the approach to fine-tuning in one of our experiments. The next important design decision is that we do not work on the software models directly but—as previously—we work on the simple change graphs from Definition 2.4.4. The basic idea is that simple change graph completions can be straight forwardly interpreted as model completions (i.e., generating a new "added" node corresponds to adding a new model element to the model, etc.). Working with the concept of a simple change graph has several advantages: First, we do not have to work with the entire software model representation, but we can focus on slices of the models around recently changed elements. This is one tactic of dealing with the common problem of the limited context of a large language model. For example, in our running example, the entire (serialized) UML meta-model is huge and would not fit in the context of contemporary large language models. Second, simple change graph completions also include attribute changes and deletions of model elements and are not limited to the creation of new model elements. RAMC is capable of suggesting semantically appropriate changes, such as renaming an attribute or altering the type of an attribute. Additionally, it recommends specific attribute values that are beyond predefined options, for example, values for string type attributes. Although alternative representations besides simple change graph can influence the outcome, choosing simple change graph was a deliberate design decision we made.

An overview of the approach is depicted in Figure 8.3, the computation of model differences (Figure 8.3, ①) and simple change graphs (Figure 8.3, ②) works similar to the corresponding steps in OCKHAM, with the major difference that we include all attribute information as well. Their serialization will be addressed in the next subsection. In the appendix in Section E.2, we formalize our approach based on the terminology from Section 2.4.2.

8.2.3 Pre-processing

Both training phase and generation phase work on serializations of simple change graphs. We describe how these serializations are derived based on the example given in Figure 8.1. Input to this procedure are two (successive) revisions of a model; output is a serialization of their simple change graphs. These revisions can originate either from the model the user is working on (in the generation phase) or from our training data.

In the first step, a model difference is computed for each pair of successive revisions of a model (Figure 8.3, ①). Regarding our running example in ⓐ of Figure 8.1, we also highlighted these model differences by color, that is, "added" model elements

4 Software models can become huge compared to the limited number of tokens that can be given to a large language model.

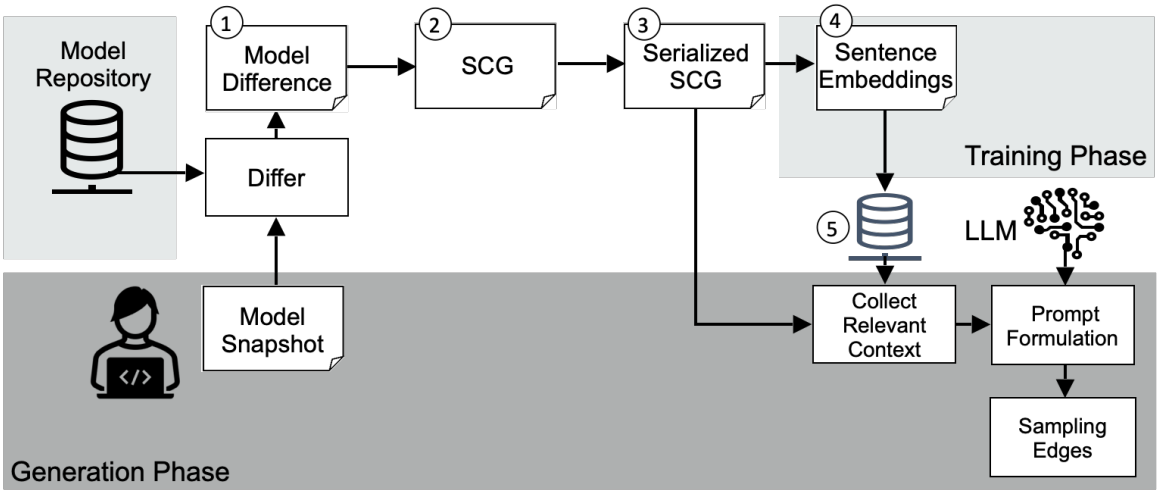

Figure 8.3: Overview of the approach RAMC. Simple change graphs will be computed and serialized. They will be retrieved via a semantic search and added to the context for the model completion task.

are depicted in green. From this model difference, we compute a (partial) simple change graph (see Definition 2.4.4 and Figure 8.3, ②). Finally, the simple change graph is serialized as a list of edges (Figure 8.3, ③). To this end, we defined a graph serialization, called *EdgeList*, for directed labeled graphs (see Section D.2). Figure 8.2 presents the prompt generated from our approach alongside the corresponding response, which was retrieved via API access to ChatGPT. It also shows an example of this graph serialization (e.g., last part of the prompt), which contains all kinds of attribute information. It can quickly become verbose and noisy in real-world examples. Common formats such as the GraphML[5] are less suitable for large language models, since they list vertices before edges. This requires guessing all nodes first—added, deleted, and preserved—before generating edges.

8.2.4 Training Phase

The *input* to the training phase is a set of serialized simple change graph components. The *output* is a (vector) store of serializations with a key for retrieval (Figure 8.3, ⑤). We retrieve relevant simple change graphs from model repositories by utilizing a *similarity search* based on sentence embeddings [264]. The serializations are stored in a vector database together with their sentence embedding (Figure 8.3, ④ and ⑤).

5 http://graphml.graphdrawing.org

8.2.5 Generation Phase

The *input* to the generation phase is a set of serialized simple change graph components capturing the difference of a new model snapshot (i.e., local changes) and the previous model revision ($m_1 \xrightarrow{\varepsilon} m_2$), as well as the vector store from the training phase. The *output* is a (list of) completion(s) in the form of EdgeList serializations, which are suggested to the user after being parsed (an example is given in Figure 8.2, at the bottom under 'Response').

Retrieval. The vector store is queried for simple change graph serializations via a similarity-based retrieval. Note that, in our case the retrieved context can be interpreted as *few-shot examples*, because we retrieve complete simple change graphs, that is, completed partial simple change graphs from the history. The few-shot samples from Figure 8.2 are detailed in Section E.5 in the appendix. To ensure a diversity of samples, we use a procedure similar to maximum marginal relevance [52], also explained in detail in Section E.3.3. As few-shot samples, we select up to 12 serialized simple change graphs; we investigate the dependency on the number of few-shot samples in Section 8.3.

Prompt formulation. The prompt (input to the large language model) used by our approach consists of an instruction at the beginning, followed by the few-shot samples retrieved from the vector store (joined via a separation token), and finally the (partial)-simple change graph serialization is concatenated (see Figure 8.2).

Sampling new edges. We can sample multiple completion candidates from the large language model by using a beam search or by instructing the large language model to generate multiple edges. Details of the edge sampling are given in Section E.3.4 in the appendix.

8.2.6 Implementation

We have implemented the computation of model differences and simple change graphs on top of the ECLIPSE MODELING FRAMEWORK [310], using SIDIFF [284] for matching and diffing. The other parts are implemented in PYTHON3, mainly utilizing NETWORKX[6] for handling graphs. We use LANGCHAIN[7] for the handling of language models and retrieval-augmented generation. We use the ALL-MINILM-L6-V2[8] language model for the sentence embeddings since it performed well in preliminary experiments. As vector store, we use CHROMADB[9]. As language model, we use GPT-4 (version 0613), since it performed best in preliminary experiments. We use a dedicated deployment of OpenAI on Microsoft Azure that is certified for the classification level of the industrial data.

6 https://networkx.org
7 https://python.langchain.com
8 https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2
9 https://www.trychroma.com

8.3 Evaluation

The purpose of this chapter is to evaluate if and to what extent large language models can build the basis for a suitable generative model for supporting software model evolution. For this, we need to understand the merits of large language model technology for model completion and understand several alternative methods to apply the technology for this task, including a comparison to other approaches. We evaluate to what extent our approach is able to derive structurally and semantically correct completion operations from the software model history. This includes, in particular, their applicability in industrial scenarios. We aim at a systematic evaluation of large language models for model completion in a controlled setting. This allows us to concentrate on the core effectiveness of large language model technology, while controlling for confounding factors such as tool use and human aspects (e.g., UX design facets). This is also the reason why, at this stage, conducting a user study settled in a specific application context would come with disadvantages and needs to follow at a later stage. However, by applying our approach to a real-world context at our industry partner, who expressed clear interest in and demand for this technology, we establish a solid methodological and empirical foundation, before considering the development of sophisticated and potentially costly tools.

8.3.1 Research Questions for Experimental Context

To understand the merits of language models for model completion, we want to answer the following research questions:

> **RQ 1:** *To what extent can pre-trained language models and retrieval-augmented generation be used for the completion of software models?*

Clearly, a general pre-trained language model is typically not aware of the syntax and domain-specific semantics of the simple change graph serializations *per se*. This includes the definition of the graph serialization format, the definition of simple change graphs, the meta-model, and the domain-specific semantics of the software models not already encoded in the meta-model. For example, a generated completion might be invalid according to the meta-model, (e.g., invalid combination of edge, source, and target node labels) or could even result in an invalid directed labeled graph serialization (e.g., they do not adhere to the EdgeList format).

> **RQ 2:** *How does* RAMC *compare to other approaches?*

As motivated in Section 8.2, providing context that is semantically close to a to-be-completed change could improve the correctness of retrieval-augmented generation.

We therefore want to understand the influence of the similarity-based retrieval on model completion. That is, we want to compare semantic retrieval and random retrieval of few-shot examples and to analyze the influence of the number of few-shot examples. We also evaluate the accuracy of our proposed approach, RAMC, by comparing it to the closely related work of Chaaben et al. [58], which we use as a baseline. Their study focuses on few-shot learning to suggest new model elements, providing the same unrelated, few-shot examples independently of the current model to be completed. Our investigation centers on the prediction improvements that can be realized by providing semantically similar examples from the model history as context to the large language model for the model completion task.

RQ 3: *What are limitations of using large language models for model completion in a real-world setting?*

While quantitative results provide insights into the merits of large language models on model completion, we also want to investigate when and why model completion fails. From simple examples and simulated changes it is hardly possible to make assertions for real-world changes. We therefore take a closer look at a sample set of *real-world changes*. From our observations, we will derive research gaps and hypotheses for future research.

RQ 4: *What insights can be gained when comparing domain-specific fine-tuning to our retrieval-based approach* RAMC*?*

An alternative to retrieval-augmented generation is domain-specific fine-tuning. We explore its viability, considering dataset properties and training specifics (e.g., epochs and base large language model).

8.3.2 Datasets

To answer our research questions, we make use of all three datasets from Chapter 5. Due to language model usage cost, we will usually sample from these datasets and not perform the experiments on every sample in the dataset. We will describe the sampling in Section E.1 in the appendix.

8.3.3 Operationalization

We conduct 4 experiments, one per research question. For all significance tests, we use a significance level of $\alpha = 0.05$.

Experiment 1 (RQ 1): To answer RQ 1, we preprocess all three datasets from Section 8.3.2 and generate a collection with training (75%) and testing samples (25%), more specifically simple change graphs, to ensure a systematic evaluation. We then select between 122 and 221 samples, depending on the dataset from the testing set and, for each, we select between 1 to 12 few-shot samples from the training set. The reason to choose between 122 and 221 samples is (1) to obtain a sample set of a manageable size that we can manually analyze and that induces acceptable costs for the large language model usage and (2) to obtain a large enough set to draw conclusions.

First, we analyze the correctness of the generated completions with respect to the ground truth. A simple change graph contains a change that actually occurred in the modeling history. From the change graph, we randomly remove edges to obtain a partial change graph, with the full change graph being the corresponding ground truth. This approach improves over previous methods that involves arbitrarily removing elements from a static snapshot. By focusing on model histories, we create a realistic setting, selecting subsets of changes that have actually occurred in real-world scenarios. We consider different levels of correctness: *Structural correctness* ensures that the graph structure is correct, with properly directed, sourced, and targeted nodes. *Change structure correctness* builds on this by additionally requiring correct types of changes to the model, such as whether elements should be modified, added, or removed. Lastly, *type structure correctness* demands further an exactly correct 'type' and 'changetype'. An illustrative example for these types is given in Figure 8.2 under 'response'. We automatically check the format, structural correctness, change semantics, and type correctness for all datasets.

For the INDUSTRY dataset, we additionally manually evaluate the generated completions for *semantic* correctness. In our manual analysis of *semantic correctness*, a solution was deemed correct if the large language model's proposed completion matched the ground truth in meaning and purpose. This check cannot be automated due to the extensive use of natural language in our data and application-specific identifiers (e.g., user-chosen attribute names). For example, in Figure 8.2, naming a new operation 'getExtension' or 'getExt' is a matter of preference, while their semantic meaning is the same. We addressed potential errors and bias in our manual analysis by having two of the authors independently evaluate the proposed solutions. Any mismatches in their evaluations were discussed, and a consensus was reached on the correct interpretation. For the base large language model, we use GPT-4[10] (version 0613) in a dedicated Azure deployment to complete our prompts.

10 Note that we experimented with several large language models from the GPT family of models and also observed changes in the specific model's performance over time [61]. At the time of execution, GPT-4 using a small introductory prompt that explains the tasks (see Section E.3.4) was performing best on a small test set, and we therefore fixed the large language model in RAMC to GPT-4.

Experiment 2 (RQ 2): In RQ 2, we investigate whether the correctness (from correct format to semantic correctness) depends on the number of few-shot samples. For the INDUSTRY dataset, we have the information on whether a few-shot sample's change is of a similar class as the test simple change graph. We also investigate how this affects correctness, that is, whether the similarity-based retrieval in RAMC affects the correctness of completions. To this end, we compare semantic sampling with few-shot samples that have been randomly retrieved from the training data. We evaluate this for semantic correctness. For this reason, and also to reduce the large language models usage costs, we perform this analysis only for the INDUSTRY dataset.

Furthermore, to compare RAMC to a baseline, we chose the model completion approach by Chaaben et al. [58]. We selected the publicly available REVISION dataset for comparison. This choice not only enhances reproducibility but also allows for comparisons with future methodologies, such that ongoing research advancements can be directly compared to our RAMC and the work by Chaaben et al. [58]. Their approach recommends new classes, their associations, and attributes. Accordingly, the present experiment specifically targets these aspects. We excluded samples that did not fall into these categories, resulting in 51 test examples from the REVISION dataset for comparison. To replicate the approach introduced by Chaaben et al., which we denote by BASELINE, we use their few-shot examples, serialization of concepts, and incorporate the partial models similarly into the prompt. We query GPT-3 (text-davinci-002) several times and suggest the most frequently occurring concept. Further details of the adaption of their approach are available in Section E.4, in the appendix.

Experiment 3 (RQ 3): We answer RQ 3 by manually investigating completions that have been generated in the first experiment for the INDUSTRY dataset. We go through all prompt and completion pairs and identify common patterns where the model completion works well or does not, and we aim at interfering causes that led to the results. Since this analysis is time-consuming, we focus on the INDUSTRY dataset—a domain- and project-specific, real-world dataset. We report on the identified strengths and weaknesses of the approach—given this real-world scenario—and point to research gaps and formulate hypotheses for future research and improvements.

Experiment 4 (RQ 4): To investigate whether fine-tuning is a viable alternative to few-shot prompting (see Experiment 1), we fine-tune models from the GPT family of language models on the SYNTHETIC dataset. The reasons why we restrict this analysis to the SYNTHETIC datasets are manifold: The main reason is that we want to understand *how* the performance of the fine-tuning approach depends on various properties of the dataset in a controlled setting. Furthermore, we have a limited budget for this experiment, and fine-tuning is costly. We also control for the number of fine-tuning epochs and the base language model used for the fine-tuning. For

every repository of the dataset, we split the data into training set (90%) and testing set (10%), and we use the test set to report on the performance of the completion task. The fine-tuning of the models optimizes the average token accuracy[11]. To compare the retrieval-augmented generation to fine-tuning, we run both for the same test samples. For the few-shot training samples, we also use the same training samples used to fine-tune the language models. We assess the correctness with regard to the ground truth. Due to the unique characteristics of the SYNTHETIC dataset, the ground truth correctness is defined by the graph structure, change structure, and type structure, that is, a manual check for semantic correctness is not necessary.

8.3.4 Results

Experiment 1 (RQ 1): Addressing RQ 1, which explores the extent to which pre-trained large language models and retrieval-augmented generation can be utilized for software model completion, our findings on the correctness of RAMC are detailed in Table 8.1.

We list the different *levels of correctness* for all datasets. We see that more than 90% of the completions have a correct format and even more than 76% of completions are type correct, that is, completed edges have the right source and target nodes, and edge label and the labels of the source and target node are correct. Even at a semantic level, 62% of the generated completions are correct for the INDUSTRY dataset. For the SYNTHETIC dataset type correctness is equivalent to semantic correctness. Consequently 86% of the results are correct for this dataset.

Table 8.1: Different levels of correctness in percent (%) of the entire test set for all three datasets.

Dataset	Format	Structure	Change Structure	Type Structure	Semantic	Total Count
INDUSTRY	92.62	86.89	78.69	76.23	62.30	122
REPAIRVISION	91.86	84.62	84.16	76.92	–	221
SYNTHETIC	99.05	86.19	86.19	86.19	–	210

Experiment 2 (RQ 2): Regarding the relationship between the number of few-shot samples and correctness, we conducted a (one-sided) Mann-Whitney-U test for the overall and type/semantic correct distributions over the number few-shot samples.

11 At the time of experiment execution, evaluating with any self-defined test metrics was not possible using the fine-tuning APIs provided by OpenAI. This metric is not aware of any specifics of the dataset, and even a single wrong token in a serialization can produce a syntactically wrong serialization, while the token accuracy for the incorrect completion would still be high.

For every dataset, we do not find any significant relationship between the number of few-shot samples and correctness (smallest p-value is 0.2 for the type correctness of the REPAIRVISION dataset). Furthermore, we find that test samples where a similar class of changes is among the few-shot samples perform significantly better than overall correctness ($p = 0.0289$ using a Mann-Whitney-U test, $p = 0.0227$ using a binomial test). Finally, we find that similarity-based retrieval performs significantly better than random retrieval for type correctness ($p < 10^{-9}$, using a binomial test) as well as for semantic correctness ($p < 0.0038$ by a binomial test[12]).

Table 8.2: Different levels of correctness in percent (%) of RAMC and random retrieval on the INDUSTRY dataset.

Approach	Format	Structure	Change Structure	Type Structure	Semantic	Total (Count)
RAMC	92.62	86.89	78.69	76.23	62.30	122
RANDOM	84.43	79.51	52.46	50.00	–	122

To obtain a clear picture of the pros and cons of RAMC, BASELINE and random retrieval, we independently report the accuracy of the correct concepts (classes) and the correct association. We further split correct concepts in correct type ("Same Class" in Table 8.3) and correct name (see Section E.4 for details). We perform binomial tests (our random baseline against RAMC and Chaaben et al.) to compare the effectiveness of our approach. In all cases, RAMC performs significantly better than RANDOM, which, in turn, performs significantly better than BASELINE (Table 8.3).

Table 8.3: Different levels of correctness (%) of RAMC, random retrieval, and BASELINE on the REVISION dataset.

Approach	Same Class	Same Name	Same Concept	Same Assoc.
RAMC	94.1^{**}	96.1^{**}	94.1^{**}	80.4^{*}
RANDOM	78.4	80.4	76.5	68.6
BASELINE	21.6^{**}	9.8^{**}	9.8^{**}	7.8^{**}

(**: $p < 0.01$, *: $p < 0.05$)

Experiment 3 (RQ 3): To better understand when and why the retrieval-augmented generation succeeds or fails when completing software models, we separate our

12 For semantic correctness, we rely on the fact that the number of semantically correct samples is smaller than the number of type correct samples. Thus, we are able to compute an upper bound for the p-value using the type correct random retrieval samples.

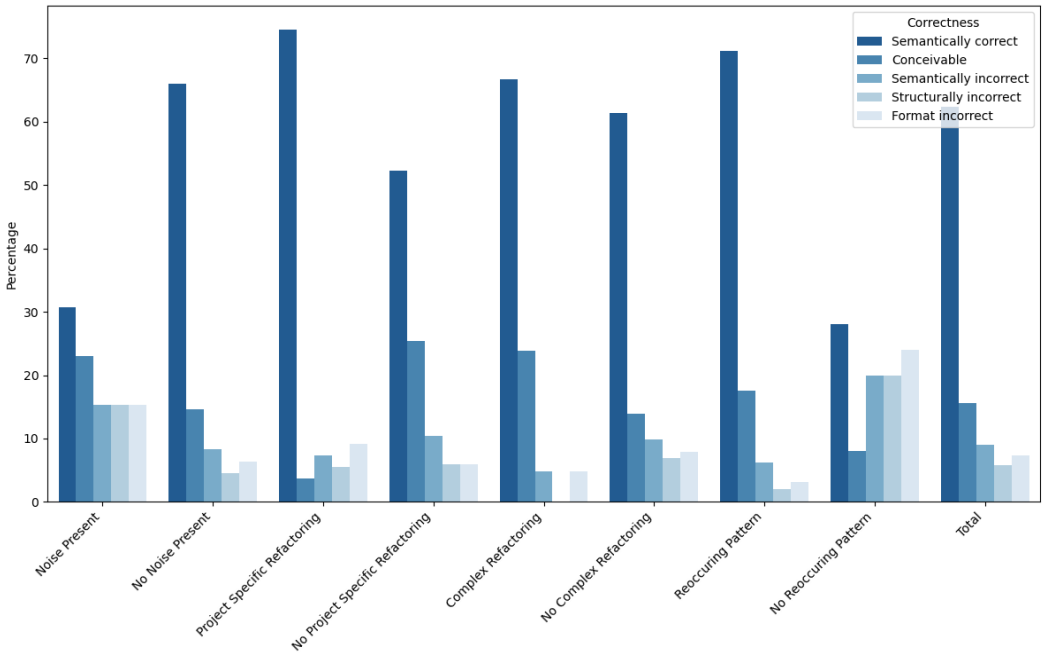

Figure 8.4: This plot shows the distribution of correctness for different categories such as presence of noise in the samples, or whether the sample represents a complex refactoring.

analysis here in two parts—successful completions and unsuccessful ones. Figure 8.4 shows a bar blot of the percentage of completions according to the different levels of correctness, separated by the different categories of completions.

Reoccurring patterns (success): Several of the successful completions follow repeating completion patterns. For example, there is a move refactoring, where a package declaration with type definitions is moved from one package to another package. Since this happened quite often in the past repository histories, the correct new parent package could be deduced, even though this package is not yet part of the incomplete test sample.

Complex refactorings (success): Furthermore, more complex refactorings have also been be completed correctly, for example, a redesign of a whole-part decomposition including packages and SysML block definitions has been correctly performed. Similarly, we find correctly completed refactorings dealing with inheritance (of port types).

Project-specific concepts (success): Even project-specific concepts, such as a special kind of tagging concept to mark software components as "frozen", are correctly inferred from the few-shot examples or co-changes of components are correctly identified, likewise.

No memorization (success): We also observe correct handling of structure in non-trivial cases. For example, correct combinations of source and target node ids are generated can not be observed in the few-shot examples.

Noise (success): The language model is able to infer concepts among noise, that is, among unrelated changes. For example, there are correctly completed instances of the "add interface block and type reference" concept where similar few-shot samples are only present with lots of entangled changes.

Regarding unsuccessful cases, we observe two main reasons for failure: incorrect structure and incorrect semantic.

Structural conflicts (failure): For incorrect structure, we find examples where conflicts occur because a node with the same node id is already present. Furthermore, sometimes (correct) model elements or packages are added to the incorrect parent package (in most cases, we see a tendency of the large language model to "flatten" hierarchies).

Structure incorrect (failure): There are several instances where correct edge, source, and target node types are generated but their ids, and consequently the structure, is incorrect.

Semantics wrong b/c copy&paste (failure): One cause for incorrect semantic completions is that parts of few-shot samples are incorrectly copied and pasted. This typically occurs when the large language model lacks sufficient context to generate the correct completion, leading it to mistakenly copy and paste segments from the provided examples.

Semantics wrong b/c unknown evolution/missing context (failure): For example, in the case of functional project-specific evolution, it might be hard to "guess" the right completion without further knowledge, or the semantic retrieval might fail to retrieve instances of the correct change pattern. Interestingly, in some of these cases, the large language model is "guessing well but not perfect" (e.g., added subsystem instead of external subsystem).

Conceivable but unobserved evolution (failure): Another interesting instance of incorrect semantic completion is a completion where a comment (in German) should be removed but instead a comment (in English) has been added. In the project, there were many renamings from German to English and, in this case, a future change has been correctly anticipated.

Experiment 4 (RQ 4): To compare our retrieval-augmented generation-based approach to fine-tuning, we perform an analysis at the token level, and we also compare the completions on a graph-structural and semantic level. At the token level, we find an average token accuracy of 96.9%, with a minimum of 92.1%, and a maximum

of 99.0% on our test data sets (10% test ratio). We can observe strong correlation of the average token accuracy with the number of fine-tuning epochs. Also, "larger" language models perform better with respect to the average token accuracy. Regarding the repository properties, we only find significant negative correlations with the perturbation probability. That is, more diverse repositories are typically harder for the model completion using fine-tuning. Exact numbers are given in Section E.8. When comparing the distributions of the edges removed in the simple change graph for incorrect and correct completions, we see that the average number of removed edges for the incorrect (i.e., no exact match) completions (5.78) is significantly larger than the average number of removed edges for the correct ones (2.94). Similarly, we find a significant relationship for the distributions of the total simple change graph size (14.89 for the incorrect completions, and 6.39 for correct completions). Accuracies of the comparison of our approach to the fine-tuning approach are given in Table 8.4.

Table 8.4: Different levels of correctness in percent (%) for fine-tuned models compared to the retrieval-based approach in multi-edge software model completion on SYNTHETIC.

Dataset	Method	Correct edge(s)	Exact match
SYNTHETICSAMPLEADA	RAMC	88.52	39.34
	text-ada-001	88.33	56.67
SYNTHETICSAMPLECURIE	RAMC	86.00	37.00
	text-curie-001	90.05	64.68

We conducted a Mann-Whitney-U test to compare the performance of retrieval-augmented generation and the fine-tuned text-curie-001 and text-ada-001 models from the GPT-3 family. In terms of producing, at least, one correct edge, neither fine-tuning nor retrieval-augmented generation exhibit statistical significance in outperforming the other. In terms of exact matches, text-ada-001 ($p = 0.0290$) and text-curie-001 ($p < 10^{-7}$) outperform retrieval-augmented generation. Regarding exact matches, the impact of different sampling methods used in fine-tuning and RAMC becomes substantial (algorithms are provided in Section E.3.4, in the appendix). While RAMC often produces more edges than required, the sampling procedure used with the fine-tuning models is more conservative.

8.3.5 Discussion

Overall, we find that both RAMC and fine-tuning of large language models are promising approaches for model completion, and the general inference capabilities of large language models are useful, can handle noisy contexts, and provide real-time capabilities. We will next discuss the results, outline hypotheses for potential future research, and describe threats to validity in Section 8.3.6.

RQ 1: *To what extent can pre-trained language models and retrieval-augmented generation be used for the completion of software models?* In Experiment 1, we observed promising correctness values across all datasets. Not only are more than 90% of completions correct w.r.t. the serialization format, but we also find *more than 62% of semantically correct completions in the real-world industrial setting*.

This indicates that retrieval-augmented generation is a promising technique for model completion. Token processing times fall within the millisecond range, and time required for semantic retrieval is negligible, even for larger models. The approach's real-time capability is significant given the stepwise model completion use case.

RQ 2: *How does* RAMC *compare to other approaches?* In the retrieval-augmented generation setting, we do not find any significant relationship between the number of few-shot samples and correctness. We find that similarity-based retrieval boosts the correctness of the approach and that it significantly performs better if a similar relevant change—following a similar pattern—is available in the context. We have observed that, in all instances where new elements with associations are recommended, RAMC consistently outperforms random retrieval and the baseline by Chaaben et al. [58].

These results reinforce our findings from RQ 1, namely that leveraging large language models with retrieval-augmented generation represents a viable approach for model completion. It is also worthwhile mentioning that real-world datasets are typically biased with respect to the change pattern (see 6.4.2), and semantic retrieval can avoid sampling from large but irrelevant change pattern.

RQ 3: *What are limitations of using large language models for model completion in a real-world setting?*

We have seen that our approach can be used to provide completions that are correct to a large extent for simple reoccurring patterns but also more complex refactorings. Even project-specific concepts can be deduced from few-shot examples. In many cases, generated edges are also structurally correct. The general inference capabilities of large language models are useful, for example, in dealing with concepts for which there are few or no similar examples. Furthermore, also in the presence of noise, retrieval-augmented generation often provides correct completions. Regarding usefulness of the completions, our manual analysis reveals that many of the completions appear useful for the modeler. For example, RAMC was able to perform a translation of several German comments to English, because the engineering language of the project has been changed. Furthermore, RAMC was able to complete project-specific refactorings. For a further investigation of these observations, we formulate the following hypothesis.

Hypothesis: Large language models and retrieval-augmented generation are able to handle noisy training examples, leverage (domain) knowledge from pre-training, adapt to project-specific concepts, and provide useful software model completions.

We found completions that are incorrect from a structural viewpoint as well as incorrect from a semantic viewpoint. As for structurally incorrect completions, we identified cases where existing node ids are incorrectly reused, where incorrect (containment) hierarchies would have been created, or where completed edges are correct from a type perspective but do not connect the right nodes. It is worth further investigating how these structural deficiencies could be overcome, in particular, given that large language models are designed for sequential input, not for graph inputs. This leaves us with the following hypothesis.

Hypothesis: Conceivable remedies for the structural deficiencies include fine-tuning of large language models, combining graph neural networks—designed for graph-like input—with large language models, providing multiple different graph serialization orders, or a positional encoding that reflects the graph-like nature of the simple change graph serializations.

Regarding semantics, we found incorrect completions that were related to a lack of (domain) knowledge in the pre-trained model or the few-shot examples, respectively. For example, we found cases of functional evolution where the language model is missing (domain) knowledge or requirements, or cases of a refactoring without any relevant few-shot sample. We further identified cases where a conceivable completion has been generated but was not the one from the ground truth.

Hypothesis: Conceivable remedies for the semantic deficiencies include strategies to further fuse the approach with context knowledge (e.g., fine-tuning, providing requirements, or task context in the prompt, leveraging other project data in repositories etc.). Furthermore, providing a list of recommendations may cure some identified deficiencies.

This means, large language models constitute a suitable base technology. RAMC can be extended, by adding a task description to the context or integrating other kinds of helpful knowledge into the prompt.

RQ 4: *What insights can be gained when comparing domain-specific fine-tuning to our retrieval-based approach* RAMC*?* We found that more fine-tuning epochs are beneficial for the average token accuracy. More diverse repositories increase the difficulty for the software model completion. The larger the simple change graph and the more edges we omit for the completion, the higher the probability of an incorrect completion. Regarding exact matches, fine-tuning—with its particular beam-like sampling—performes better than RAMC.

The reason that fine-tuning has a higher exact match accuracy is more due to the edge sampling algorithm than to the method itself: When analyzing the percentage of correct edges, it becomes clear that we cannot conclude that one approach outperforms the other. Instead, we hypothesize a strong dependency on the edge sampling procedure, which deserves further investigation. While the retrieval-augmented generation often generated more edges than necessary, the sampling procedure used with the fine-tuned models from the GPT-3 family takes a more conservative approach, prioritizing the generation of edges with high confidence.

Comparison to code completion. Note that large language models for source code completions show similar results to our findings in Experiment 1 and 4, ranging from 29% for perfect prediction of entire code blocks to 69% for a few tokens in a single code statement [68]. Drawing a direct comparison between code and model completion is not straightforward, though.

8.3.6 Threats to Validity

With respect to construct validity, we made several design choices that may not be able to leverage the entire potential of large language models for software model completion, including our definition of simple change graphs, the serialization of the simple change graph, the strategy of how to provide domain knowledge to the language model, and the choice of the base large language model.

To increase internal validity, we incorporated the SYNTHETIC Ecore Dataset into our experiments, controlling for properties of software model repositories. Still, we were not always able to completely isolate every factor in our experiments. For example, fine-tuning and few-shot learning use different edge samplings. This is due to the different interfaces (completion interface for fine-tuning, and chat-like for retrieval augmented generation) that we used to access the language models. In future research, an ablation study for the design choices in the algorithms shall be performed. To address the potential variability that large language models may exhibit, we checked and confirmed that the completions were stable.

Regarding external validity, we included two real-world datasets (REPAIRVISION and INDUSTRY), and we study real-world change scenarios from the observed history in these repositories. We have chosen our test samples to be small enough to perform manual semantic analysis, but large enough to draw conclusions. To minimize costly manual checks, our semantic analysis was confined to our most challenging dataset, the INDUSTRY dataset. Extending the analysis to the other datasets would enhance validity. However, we are confident that, having analyzed hundreds of samples, we have struck a reasonable compromise. We are therefore certain that our results have an acceptable degree of generalizability for the current state of research. In any case, user studies shall investigate the usefulness of our completions in practice. Investigating merits of large language models for model completion is an emerging topic, and many questions are open. Still, our results set a lower bound for the

potential of large language models in this area, with promising results, insights, and hypotheses for further research.

8.4 Related Work

In this section we will review the literature in the area of software model completion and related areas. A table summarizing and comparing related work to our approach is given in the appendix (cf. Section E.9). In particular, in the appendix we will discuss why a direct comparison of different approaches to model completion is difficult, in general.

Various approaches have been proposed for software model completion, ranging from rule-based approaches to data mining techniques and more sophisticated machine learning approaches. An overview of recommender systems in Model-driven Engineering is given by Almonte et al. [14]. Previous work studies recommending model completions by utilizing knowledge bases such as pattern catalogs or knowledge graphs [7, 79, 193, 194, 205, 217, 236]. Consequently, these research efforts are often domain-specific, as they require the provision of domain-specific catalogs (sometimes referred to as the cold start problem), such as for UML [193, 194, 236] or business process modeling [79, 205].

Another common approach is to use already existing model repositories and employ techniques such as frequency-based mining, association rule mining, information retrieval techniques, and clustering to suggest new items to be included in the model [5, 80, 92, 312] or new libraries for use [129]. MemoRec [80] and MORGAN [82] are frameworks that use a graph-based representation of models and a similarity-based information retrieval mechanism to retrieve relevant items (such as classes) from a database of modeling projects. However, their graph-based representation focuses on the relationship between a model element and its attributes, but it does not capture relationships *between* different elements in the model and consequently may not capture the essential semantics and constraints of the model and modeling languages. Repository mining and similarity-based item recommendation techniques are often combined [79, 205]. Kögel et al. [179, 181] identify rule applications in current user updates and find similar ones in the model's history. More generally, one could automatically compute consistency-preserving rules [166] or pattern mining approaches [197, 323, 324] such as OCKHAM from Chapter 6 to derive a set of rules to be used in conjunction with a similar association rule mining approach. As discussed in Section 8.1, these approaches will struggle to sufficiently cover the long-tail of context-specific edits to a model.

Another strategy to generate model completion candidates that comply with the given meta-model and additional constraints involves using search-based techniques [309]. Without knowledge about higher-level semantics, these approaches are

more comparable to the application of a catalog of minimal consistency-preserving edit operations [166].

Regarding the application of natural language processing (NLP) [48] and language models [67, 345], Burgueño et al. [48] propose an NLP-based system using word embedding similarity to recommend domain concepts. Weyssow et al. [345] use a transformer-based language model to recommend meta-model concepts without generating full model completions. Di Rocco et al. [81] introduce a recommender system using an encoder-decoder neural network to assist modelers with editing operations. It suggests element types to add, but leaves the specification of details, values, and names of these elements and operations to the human modeler. Gomes et al. [114] use natural language processing to translate user intents, expressed in natural language, into actionable commands for developing and updating a system domain model. While code completion and model completion are closely related, recent research has mainly concentrated on code completion, where LLMs seem to be the state of the art [62, 68, 150, 305]. Considering the close connection to code and model completion, it's essential for us to explore further how generative approaches, such as LLMs, operate within the context of software model completion of complex real-world models. Most closely to this work, is an approach by Chaaben et al. [58], which utilized the few-shot capabilities of GPT-3 for model completion by providing example concepts of unrelated domains. In contrast, our approach takes a different avenue, leveraging model evolution from model repositories. Cámara et al. [51] further extend on Chaaben et al.'s research by conducting experiments to assess ChatGPT's capability in model generation. Ahmad et al. explore the role of ChatGPT in collaborative architecting through a case study focused on defining Architectural Significant Requirements (ASRs) and their translation into UML [9].

A slightly different but similar research area focuses on model repair [142, 218, 237, 240, 244, 312]. REVISION [244]—from which we have also taken the REPAIRVISION dataset—uses so-called consistency-preserving edit operations to detected inconsistencies and then uses the pre-defined edit operations to recommend repair operations.

8.5 Conclusion

To validate our theory from Section 3.5, in this chapter, we wanted to understand to what extent existing technology, namely pre-trained large language models with retrieval-augmented generation and fine-tuning, can support the evolution of models and provide context-specific suggestions of how to change software models.

To this end, we presented and investigated an approach to software model completion based on retrieval-augmented generation, RAMC, and compared it to fine-tuning during our evaluation.

Our experiments on all three datasets—the SYNTHETIC, the Ecore REVISION, and the SysML INDUSTRY datasets—provide evidence that, indeed, large language models are a promising technology to realize a generative model for software model evolution as in Hypothesis 2 in Section 3.5. Indeed, the technology is also able to adapt to an arbitrary specific context and therefore support modelers in many situations.

The real-time capability of our approach is especially beneficial for stepwise model completion, highlighting its practical utility. This also supports Hypothesis 1, regarding the practical utility of a data-driven intelligent modeling assistant in Model-driven Engineering. We achieved a semantic correctness in a real-world industry setting of 62.30%, which is comparable to earlier results with large language models for source code completion. Further investigation revealed that similarity-based retrieval significantly enhances the correctness of model completions and that fine-tuning is a viable alternative to retrieval-augmented generation. All in all, the general inference capabilities of large language models are beneficial, particularly in dealing with concepts for which only scarce or even no analogous examples are provided. We have identified concrete causes for the technology to fail and formulated corresponding hypotheses for future research.

Of utmost importance for future research is to compare technology, such as graph neural networks, that has been designed for processing graph-like data (e.g., our simple change graphs), especially for structural aspects of software model completion. Also, investigating hybrid approaches by marrying technologies that are strong for structural aspects, and large language models, that are typically strong for semantic aspects of model completion is worth further investigation. One of the key limitations of current research practice in this area is the lack of automated evaluation for semantic correctness, currently requiring a time-consuming and cumbersome manual analysis. We envision software model datasets with a defined execution semantics that would allow for automated evaluation of generative approaches. Alternatively, having surrogate measures for the semantic correctness could also be a remedy for the current situation.

9

Conclusions and Outlook

Every new beginning comes from some other beginning's end.

— Seneca

In the chapters of this part, we investigated several use cases and methods individually. We will now look at our results from the more general perspective of the thesis. Furthermore, we will draw a bigger picture and give an optimistic outlook for the future of data-driven intelligent modeling assistants.

9.1 Conclusion

In Section 1.2 we formulated three research goals for our thesis. We will recall these goals and assess if and how we achieved the corresponding goal.

Our first goal was devoted to develop a theory of software model evolution:

> **Goal 1:** *We will develop a theory relating software model evolution to software model histories.*
> Indeed, in the first part of the thesis, we developed a theory linking generative models to the evolution of software models. The generative models learn from the past model evolution (e.g., from data in software model repositories), to support future evolution of software models.

A key contribution of the theory is to make software model evolution measurable, via the complexity and probability of software models and biases that can be observed in historic data. We have shown how these biases give rise to (editing) patterns. In particular, we formulated Hypothesis 3 that the meaningful edit operations lead to compression of the software model histories and Hypothesis 2 that a parametric approach to a generative model for software model evolution captures regularities (i.e., the biases) in the evolution data. In particular, Chapter 6 provides first evidence to support Hypothesis 3, while Chapter 7 and Chapter 8 provides evidence for Hypothesis 2.

To make this theory of practical use, we had to collect and curate a dataset. We therefore formulated our second goal:

Goal 2: *We will curate an expressive model evolution dataset that includes real-world industrial models, open-source models, and simulated models.*
We collected a dataset comprising synthetic, public real-world, and industrial real-world data. The combination of synthetic and real-world data allows to make a good trade-off between internal and external validity of conclusions in experiments.

Many studies work on snapshots of models. These snapshots of models, do not contain any information real evolution of the models. Building on top of earlier work [244], we used existing as well as new datasets that include the entire history of models. This dataset is therefore well suited to analyze software model evolution.

As a third research goal, we wanted to investigate data-driven methods for several model evolution use cases:

Goal 3: *We will investigate and evaluate data-driven methods for model evolution use cases—including edit operation mining, semantic lifting of model differences, and model completion.*
In the second part of this thesis, we presented any evaluated approaches for edit operation mining, semantic lifting of model differences, and model completion. We evaluated these on synthetic as well as real-world datasets and found that leveraging historic data from the evolution of the software models is indeed a promising direction with advantages over hand-crafted tools.

In particular, Chapters 6–8 provide preliminary evidence for Working Hypothesis 1 that knowledge can be derived from data in software model repositories and leveraged to provide useful intelligent tool support to support software model evolution.

We are confident that these results are generalizable to other use cases as well. For the future, it is imperative to collect even larger datasets and extend the evaluation to new domains and new use cases not considered in this thesis (e.g., model summarization, change propagation, or model repair).

Also, hybrid approaches are a promising future direction. For example, fine-tuning models on diverse artifacts for a certain domain and then combining these fine-tuned models with retrieval-augmented generation or similar approaches will likely improve the quality of the completions. Furthermore, combining graph-mining and graph neural networks for structural aspects with large language models for semantic aspects can join the strength of both worlds for intelligent modeling assistant uses cases.

9.2 Bigger Picture and Future Work

In this thesis, we focused on concrete use cases such as edit operation mining or software model auto-completion. However, the vision for the future of modeling is much broader.

The idea of Model-driven Engineering and Model-driven Architecture is to be an abstraction layer on top of the executable code that eases communication and development of the products. Similar ideas have been pursued by low-code platforms, which aim to make software development more accessible to non-programmers.

In the past, the technologies developed in the field of Model-driven Engineering (e.g., MOF [245], model transformations [246], etc.) have been used, for example, for documentation purposes or to improve the quality and traceability of the development artifacts. Anyway, Model-driven Engineering did not bring a paradigm shift to the world of software, unlike higher-level programming languages like Fortran in 1957 almost completely removed the need to write machine code or assembly. Researcher have hypothesized in the past [50] that, roughly speaking, making the modeling tools more intelligent can change this picture.

Indeed, our research supports this hypothesis, in the sense that we see that contemporary technology has the potential to significantly reduce the perceived cost of the modeling itself. Still, the vision of completely removing the need of writing source code is far from being reached. From the side of source code, we see a similar trend to generate source code from natural language descriptions [152]. It is conceivable that in the future, these two streams will converge. That is, while we cannot expect that complex products can be completely generated out of a short natural language description[1], we can expect that a composition and more detailed description of the system will be provided in terms of models of the system, while the concrete implementation of smaller system capabilities and responsibilities will be generated. That is, the effort of clearly understanding the needs (i.e., the requirements) and the capabilities of the system will not go away but the effort of having to write source code might only be necessary in corner cases. This will also come with a shift of the effort to validation and verification of the system. The less control we have over the concrete handling of bits by the machine, the more we have to rely on the correctness of the models and therefore test the runtime behavior and the fulfillment of the desired properties and qualities of the system. Again, this is quite similar to

1 For example, the description "I want a train control system", obviously is not sufficient to deliver exactly what the customer wants or that is consistent with the existing infrastructure that the customer is operating. In other words, the solution space is still too large and includes too many possibilities that do not satisfy the customer needs. A more detailed description is necessary to constrain the solution space. On the other hand, the description "If the toilet door button is pressed, the vacancy light should switch from green to red" is already quite detailed and could be used to generate the necessary source code (given that there are corresponding variables for toilet door button and the vacancy light).

the advent of high-level programming languages, where the need for testing and verification of the source code increased.

For this vision to become reality, several challenges have to be addressed. In particular, there are many more use cases for intelligent modeling assistants that we have not covered in this thesis. In particular, the relationship between models and code needs to be analyzed in more detail. Studies investigating several aspects such as effort, quality, or efficiency of model-based development compared to code-based development are necessary.

This also requires a cultural change of how engineering tools are being developed. Currently, domain experts provide the detailed knowledge about the domain, which is then encoded by software engineers into the tools. In the future, one might expect a shift to complement the knowledge of the domain experts by knowledge extracted from existing data. In general, one might expect a more evidence-based development of tools, requiring a lot of telemetry data. This is a cultural change that is not easy to achieve, as it requires a lot of trust in the data. Additionally, engineering data is often highly confidential and tool users and tool developers are two different legal entities. It will therefore be difficult, to evaluate new or improved features with real-world data. Furthermore, the often probabilistic nature of the data-driven methods might be difficult to understand for domain experts, who are used to deterministic tools. Therefore, a certain level of explainability and transparency of the data-driven methods is necessary. This requires a strong emphasis on the verification and validation of the tools, which again requires test data as close to reality as possible.

Luckily, we see a lot of success stories in the field of data-driven software engineering, which eases and accelerates the adoption of these technologies and supports the cultural change that is required for the envisioned paradigm shift. We therefore have an optimistic outlook for the future of data-driven intelligent modeling assistants and look forward to the exciting research that is yet to come.

Appendix

A

Thesis Positioning & State of the Art of Intelligent Modeling Assistants

In this chapter, we will reflect on the state of the art of research on intelligent modeling assistants in Model-driven Engineering.[1] Furthermore, we will discuss the positioning of this thesis in the field of Model-driven Engineering recommender system research.

A.1 Positioning of this Thesis

We will discuss the application of our approaches along several purposes[2] and artifact types as depicted in Figure A.1. The terminology and taxonomy used in this figure is based on a literature review about modeling recommender systems [14] and has been slightly adapted for this thesis.

Create. We do not study model generation directly. Nevertheless, our model completion approach could be used to generate models, for example, for fuzzing or testing purposes. Furthermore, during our study of edit operation mining (see Chapter 7), we trained a graph neural network and large language models to generate model differences. These approaches, in theory, can be studied further to generate models. In this thesis, we do not further discuss this, and we also do not discuss model generation based on natural language input. Also creating models from models (i.e., exogenous model transformation), is not further discussed in this thesis. Creating of single model elements is considered a standard tool feature and therefore also not further discussed.

1 We refrain from using citations for "negative" examples, as we do not want to discredit any other researchers in this field. In fact, the author stands on the shoulders of giants and is convinced that the research community has done a great job in the past. The challenges we describe in this section are rather a systemic problem and can be attributed to the immaturity of the field and the rather low number of researchers in the field. Indeed, our research evaluation ideas and datasets are based on infrastructure that previous research has built, and we are grateful to built upon this infrastructure.

2 We also use the term *purpose* here as in Almonte et al.'s literature review. In this context, we use three terms more or less interchangeably: use case, activity, and purpose.

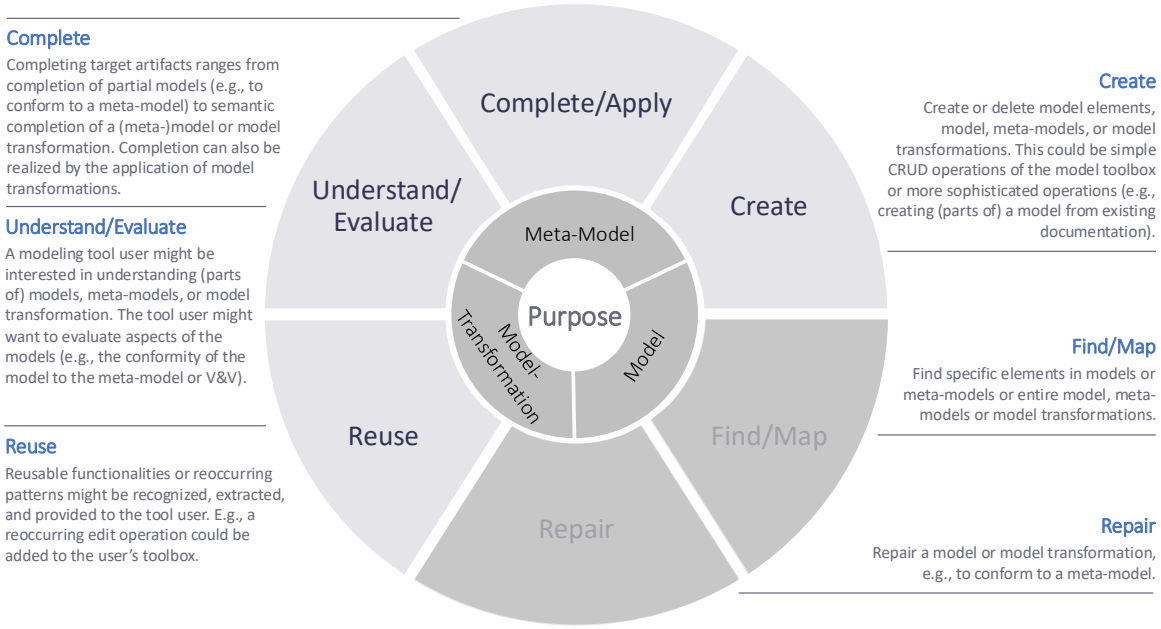

Figure A.1: Dimensions for software modeling activities from a mapping study by Almonte et al. [14]. The grayed out dimensions, *Find/Map* and *Repair*, are not further discussed or only slightly touched in this thesis.

Reuse. Feature Extraction is a form of managed reuse in product lines. Our edit operation mining approach can be considered to be a form of model reuse. Similarly, data-driven model completion can also be considered to be a form of reuse. In fact, data-driven generative approaches are always some form of reuse. Other, direct reuse, for example, extracting clones from a model, is not further discussed in this thesis.

Understand/Evaluate. Our study of edit operation mining, semantic lifting of model differences, and change profiling can be used to understand model evolution and model differences. Model evaluation (e.g., validation and verification) is not further discussed in this thesis.

Complete. We study model completion directly. Indeed, also our edit operation mining approach can be used to complete models, for example, by using the approach of Kögel et al. [180]. We do not directly discuss the application of model transformations.

Regarding artifact types, several of our approaches to use cases that we will study in this thesis—such as, edit operation mining and model auto-completion—can, theoretically, be applied to models, meta-models, and model transformations, because all can follow a certain meta-model, and our proposed approaches work

on the meta-model level—as we will see in Chapter 6–8. Therefore, the artifact type our use case is about will only depend on whether the input to our approach will be models, meta-models, or model transformations. Nevertheless, we did evaluate edit operation mining and model auto-completion only by applying it to models and meta-models.

Furthermore, in this thesis, we usually do not make strong assumptions on the models. We mainly work with labeled, directed graphs. Therefore, most of what we do can be applied to a wide variety of modeling languages and domains. Of course, this implies that we cannot leverage domain or language-specific properties in our approaches. Nevertheless, we are convinced that this more abstract research approach perfectly fits the current state of the art of the research in the field of Model-driven Engineering, that is, we first have to understand several use cases on the more abstract level, before we can think of concrete optimizations or improvements of the approaches in a specific domain or modeling language.

In general, we find that real-world, industrial modeling projects face several evolution-related challenges that could be addressed by better tooling support. We find that existing technologies from the field of AI—for example, graph mining, graph neural networks, and large language models—can be applied and are indeed promising candidates to improve modeling tools by supporting and improving several modeling activities.

A.2 State of Model-based and Model-driven Engineering Research

Recently, a study by Savary-Leblanc et al. [280] found that research on recommender systems handling graphical or graph-like artifacts (e.g., models) is rather under-represented compared to research of recommender system dealing with textual artifacts (e.g., source code). Some counterparts in the source code domain of the use cases investigated by us, such as source code completion [62], are already mature and have already made it into the industrial practice, for example, in the form of coding assistants such as GitHub CoPilot.

On the other hand, many recommender solutions in Model-driven Engineering are still in their infancy and only available through academic research tools and have not yet made it into the industrial practice [235].[3]

3 Many of the research tools are developed in the EMF ecosystem, but are not integrated into commercial solutions such as MagicDraw or Rhapsody, as often used for industrial projects.

A.3 A Need for a Common Research Infrastructure

Currently, for many use cases in Model-driven Engineering, there is no commonly accepted evaluation metric, and no commonly accepted datasets for many use cases in Model-driven Engineering [45, 75, 209, 268]. We definitely can observe a positive trend with large collections of models being made available to the public [209]. Still, there is no dataset available, or the datasets are not used across different research groups (e.g., because they are not publicly available), making it difficult to compare different approaches.

In summary, the infrastructure necessary to conduct research in Model-driven Engineering is in its infancy, and there is a need for a shared research infrastructure.

In particular, we believe the following steps could be beneficial for development of a common research infrastructure:

- **Separation of technology from tool and application context:** To understand the merits of different technologies for key tasks in software model evolution, it is necessary to separate the technology from a specific tool integration. The evaluation in the application context is important, but there is a need to enable the evaluation of technology for a particular use case, independent of the concrete tool or application context—in a more controlled environment—before transferring it to a specific application context.

- **Evaluation on real-world data:** Even though the technology should be evaluated in a controlled environment, it is important to evaluate the technology on real-world data. This is important to understand the generalizability of the technology and to understand the potential of the technology in real-world environments that are usually more complex and data is more noisy.

- **Triangulation:** Every dataset and experiment has its limitations. Therefore, it is important to triangulate the results of different experiments and datasets to understand the generalizability of the results. Triangulation is a useful tactic to balance internal and external validity of conclusions.[302]

- **Benchmarking:** Benchmarking recommender systems is known to be difficult. Anyway, to progress the field, it is important to strive for benchmarkable approaches and results, allowing others to compare their work to the results of others.

- **KISS:** Keep it simple, stupid. One general tactic to ease the comparison is to keep the experiments (and the approach) as simple as possible and as complex as necessary. The more complex the approach, the more difficult it is to replicate the approach and to attribute improvements to its specific features.

We will detail the reflections and ideas towards a common research infrastructure in Chapter A.5 in the appendix.

A.4 Balancing Internal and External Validity

The discussion of the challenges in Model-driven Engineering research can also be seen from the viewpoint of internal and external validity. In fact, there is an immanent trade-off between internal and external validity [302]. In the context of Model-driven Engineering research, the internal validity of an approach is often evaluated on a small or artificial dataset of models, datasets which are not publicly available, or rather on surrogate datasets that can hardly mimic real-world model evolution. On the other hand, external validity is typically increased by researchers by evaluating their approach in a concrete application context as part of a tool user study. Nevertheless, for many use cases in Model-driven Engineering research, there is a gap between the research that tries to maximize internal validity and the research that tries to maximize external validity. In our evaluation(s), we try to balance the trade-off between internal and external validity in a way that fits the current state-of-the-art of the research on that specific use case.

A.5 Ideas on a Common Research Infrastructure for Research in Model-Driven Engineering

Even though positive developments have been made in the field of Model-driven Engineering, there is still a need to work on reusable datasets that can be used to evaluate different approaches and use cases in the field. Also, there is a need for commonly accepted evaluation metrics that are not coupled to the concrete approach and the concrete dataset used in the evaluation.

Reusable Datasets: The scarcity of reusable datasets [45, 75, 209, 268] for many use cases in Model-driven Engineering hinders the comparison of different approaches, which is then often reduced to a qualitative analysis. The lack of proper datasets also poses a challenge for the development and evaluation of data-driven (e.g., machine learning) approaches in Model-driven Engineering. To circumvent this lack of datasets, many authors in Model-driven Engineering research report on experiences using their approaches in a concrete application context, that is, as part of a tool. Reporting on an evaluation in a concrete application setting, again, makes it difficult to compare against the approach, though—especially if the application context or the tool is not available to the public. In consequence, the generalizability in the field is rather low.

Evaluation Metrics: There are no commonly accepted evaluation metrics and often technologies or proposed approaches are evaluated in a manner that is only applicable for the specific technology at hand. That is, the evaluation uses concepts that are not problem specific, but rather solution specific.
An Example: To give an example of this dilemma, we will consider a use case that we elaborate on in the course of this thesis: *Model completion* approaches researched in the literature range from rule-based approaches over search-based approaches to statistical and machine learning approaches and combinations of these. There are only a few datasets available that can be used to evaluate model completion. Many approaches are evaluated on a small or artificial dataset of models, datasets which are not publicly available, or rather on surrogate datasets that can hardly mimic real-world model evolution. In the concrete example of model completion, the evaluation is often performed on a dataset of model snapshots, from which elements are removed artificially. Instead, it would be more realistic to have pairs of to-be-completed models and their completed counterparts.

Only a portion of the literature reports on metrics that are independent of their specific approach and only depending on the use case (i.e., model completion). Positive examples include the work on Simulink model completion by Adhikari et al. [5] or work on "concept recommendation" by Elkamel et al. [92]. For example, a model completion approach might recommend the top-10 names for meta-model classes to be added to a meta-model and the evaluation of the approach is then only based on the number of correctly recommended names among those 10 recommendations. In consequence, there would be no natural way to compare the approach to other approaches that, for example, only recommend one name. Or in general, how to compare an approach that only recommends class names to another approach that recommends classes and how they structurally relate to existing model elements. An example for a metric that is independent of the approach is the amount of model elements (including references) that have been correctly completed. In case of multiple recommendations, either all positions of incorrect and correct recommendations have to be reported for comparison or at least a metric that takes the position of the recommendation into account (such as the mean average precision [210]). This would require a ground truth of to-be-completed and completed models, which is not available for most datasets. Even if a ground truth is available, it is not easy to define what a correct completion is. For example, if a model element is missing in the incomplete model, but the model element is not required for the model to be valid, is it a correct completion or not? Likewise, if a recommended class name is a synonym of the correct class name, is it a correct completion or not?

Note that for source code, there are commonly accepted datasets such as HumanEval [62] and evaluation metrics [62, 138] to evaluate *code* completion approaches. For example, since there is a well-defined execution semantics, the evaluation of a code completion approach can be performed by checking the correctness of the code completion in a test suite. For many models (e.g., UML, SysML, ECore, etc.), there

is no well-defined execution semantics and therefore a test approach for evaluation would not be applicable to software models, in general.

A.6 Tactics to Improve Reproducibility in Model-Driven Engineering Research

We believe, that in the field of Model-driven Engineering—thanks to many important contributions over the last decades—a research infrastructure is not too far away. The following five steps (that we also adopt for the evaluation of the approaches proposed in this thesis) are our proposal towards a more mature research infrastructure in Model-driven Engineering:

Step 1 – Isolate technology questions from tool questions: In a first step, we isolate the technology questions from the tool questions, user experience, as well as adoption questions, and evaluate our approaches on a curated dataset of model histories. Of course, tool questions are of high importance—maybe even higher importance. Anyway, we are convinced that these questions should be considered in isolation from technology questions, and, in many cases, are even independent. We can—by no means—make any statements about the value add that a technological advance brings to the user, except for the direction of the value add: technological advance improves tool advance. Given the current state of the art of Model-driven Engineering research in the use cases considered in this thesis (i.e., mainly edit operation mining, semantic lifting, and model completion), we are convinced that currently the technological advance is the bottleneck. This can be attributed (at least partially) to a lack of reproducibility in this field that has been observed by researchers in the past [14]. For example, there is no point in evaluating a model completion approach as part of a tool, if the model completion approach is not able to complete models with a satisfying accuracy in the first place. We therefore focus on this bottleneck—sidestepping tool questions.

Step 2 – Evaluate on real-world data (without tool): The second step to overcome the aforementioned challenges, is to evaluate on data from a real-world application context. Currently, many offline evaluations are rather performed on synthetic datasets [14]. It is important to mention, at this point, that our research has real-world implications and is evaluated in an application context, indeed. By applying an approach to historical data, collected from real-world applications, we can evaluate the approach in this real-world context without the need to evaluate the approach as part of a tool. Furthermore, one needs to ensure that the dataset is representative for the use case at hand. For example, to study model evolution, the dataset should include models that have evolved over time and their evolution.

Step 3 – Triangulate the evaluation: As a third step, we employ triangulation in our evaluations, wherever it is reasonable. As discussed in the concrete example

of model completion, using historical, real-world, data, it is non-trivial to assess what a *correct* recommendation is. In fact, in this concrete use case, given a certain context, there might be multiple—often even infinitely many—correct completions. *We not only want to check if a recommendation exactly matches the historical ground truth, but we also want to check if the recommendation is meaningful given the available context (i.e., model history).* That is, even a recommendation that does not exactly match the historical ground truth might be a conceivable recommendation or valid *alternative*. This usually needs some time-consuming manual investigation. We propose to overcome this challenge by firstly, triangulating the evaluation on different kind of datasets—such as real-world industrial models, open-source models, and simulated models—and, secondly, by analyzing samples of the datasets with the help of domain experts. Of course, since evaluating data with the help of domain experts is a time-consuming and expensive task, this sample set has to be small, and the results have to be extrapolated to the entire dataset. Another triangulation that can often be done is to fall back to "weaker" definitions of correctness, for example, a relaxed comparison for equality of the model completion and the ground truth, ignoring class or attribute names and only focusing of structural and type equality—therefore allowing for a more automated evaluation. Triangulating across datasets and different levels of correctness, threats to external conclusion validity of the analysis by domain experts can be largely mitigated.

Step 4 – Make it benchmarkable The fourth step is about making the research benchmarkable. Most use cases for Intelligent Modeling Assistants are recommender systems "which are notoriously difficult to evaluate offline, with some researchers claiming that this has led to a reproducibility crisis in recommender systems publications", as the Wikipedia article[4] about recommender systems mentions—devoting an entire section to reproducibility of recommender system evaluation. Since there are many degrees of freedom in how to report on the results of an analysis, even for a single use case, often incompatible reports of the results are found in the literature. In some cases, evaluation metrics are coupled to the concrete approach that is analyzed. For example, a model completion approach might utilize transformation rules and the evaluation metric might explicitly take the transformation rules into account. In this case, the evaluation metric is not applicable to other model completion approaches that do not utilize transformation rules. Unfortunately, there is no cookbook for how to make research benchmarkable and probably depends on the specific use case at hand. Nevertheless, we propose to avoid any dependencies on the concrete approach or specifics of the concrete dataset used in the evaluation and to report the results as fine-grained as possible. For example, in the use case of model completion, if one investigates a recommender approach that proposes several recommendation candidates (possibly including multiple completed model elements), one should include report on the position of the correct recommendation in the list of recommendations and the position of the incorrect recommendations in

4 https://en.wikipedia.org/wiki/Recommender_system; accessed on Sep. 21st 2024

the list of recommendations. Furthermore, one should try to also report on the results of the completion of individual model elements or for a single recommendation candidate only. If one reports on aggregated metrics, also their fine-grained metrics have to be reported. Another alleviation to this challenge would be to standardize metrics per use case. Anyway, an agreement on evaluation metrics for different model recommender systems is a rather difficult challenge of the Model-driven Engineering research community [14].

Step 5 – KISS: keep it stupid simple A last step is to keep the research simple. We refrain from using overly complex methods. Every non-trivial design decision in the researched approach constitutes a potential threat to the validity of the research. For example, when we want to investigate the merits of Large Language Model technology for model completion, one should try to not add too many additional features to the model completion approach. Otherwise, an ablation study has to be conducted to investigate which improvement can be attributed to what feature. Of course, a technology is typically not applicable to a specific use case without adaption [109], but is important to keep the adaption as simple as possible—but as complex as necessary—and be aware of any design decision that has been made.

In summary, we propose to evaluate our research on real-world data, without the need to evaluate the research as part of a tool, to triangulate the evaluation wherever it is reasonable, and to use evaluation metrics that are not coupled to the concrete approach or the concrete dataset used in the evaluation.

A.7 Internal and External Validity: An Analogy to Drug Research

As already mentioned, for any research one will have to balance internal and external validity. [302] Typically, no single experiment or dataset is able to maximize both internal and external validity of the conclusions.

Usually early research in a field tries to maximize internal validity, while later research tries to maximize external validity. We will give a simplified analogy from the very mature research field of drug development to illustrate this point:

One major research direction in drug development is on the effectiveness of a drug in a treatment of a disease. Compared to the development of a recommender system in software engineering, the disease corresponds to the software engineering use case and the drug corresponds to the technology that is investigated. New drugs are typically developed from candidates that are identified through a series of research and development activities, usually based on theories of the disease and the treatment. In this first phase, one might perform simulation studies to select candidates that are then tested in a laboratory setting. This is the phase where the internal validity of the research is maximized. In the second and the third phase, the

drug is tested in a pre-clinical, and clinical setting, respectively. Only in the clinical setting, the treatment is investigated in a realistic application context—still trying to control for confounding variables (e.g., in a controlled study with a placebo and a treatment group). Because of rather low sample size of the participants in the study, the external validity of the study is still not maximized but a good trade-off between internal and external validity is achieved. This procedure allows reducing risks, direct cost, and opportunity costs of rolling out a drug that is not effective or even harmful to the public. Only after successful passing these phases (and an approval by public authorities such as FDA in the United States), the drug is rolled out to the public and the external validity of the research is maximized by monitoring the drug in a real-world (field) setting. In this phase, there is almost no control over confounding variables.

Using this analogy, the research of the use cases and the technologies we investigate in this thesis is somewhere between the first and the third phase.

A.8 Summary

To summarize our observations about the state of model recommender systems in Model-driven Engineering research and the maturity of scientific methods in this research field, several important challenges such as lack of available datasets, reproducibility, missing real-world applicability have been recognized. Furthermore, use cases such as model generation, model completion, model repair, test generation, model evaluation, model optimization, model change analysis, etc. have been identified, defined, and discussed in the literature. Anyway, for most use cases—in particular, those that we investigate in this thesis—the research can still be considered rather immature, especially when compared to similar use cases concerning source code artifacts.

B

Endogeneous Model Transformation

In this chapter, we will introduce the formalization of (endogeneous) model transformations based on graph transformations.

B.1 Graph Transformations

A graph transformation, roughly speaking, is a transformation of one graph into another. Graph transformation will typically be defined by rules that can be applied to execute the transformation.

For this, we first need to define a graph morphism.

Definition B.1.1 Graph Morphism.
A graph morphism *between two graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ is a function $m : V_G \to V_H$ between the nodes of G to nodes of H and a compatible function $n : E_G \to E_H$ between the edges of G to edges of H that preserves the graph structure: for each edge $e = (v_1, v_2) \in E_G$ the image $(m(v_1), m(v_2)) = n(e) \in E_H$ is an edge in H.*

Ehrig et al. [89] defined the so-called double-push out construction for graph transformations:

Definition B.1.2 Double-Pushout Graph Transformation.
A graph G can be transformed *into a graph H by the* transformation rule *$r = L \xleftarrow{\iota_L} I \xrightarrow{\iota_R} R$, if there exists a graph C and graph morphisms $L \xrightarrow{m} G$ (the* match*), $R \xrightarrow{n} H$ (the* co-match*), $I \xrightarrow{\psi} C$, $C \xrightarrow{\eta_L} G$, and $C \xrightarrow{\eta_R} H$, such that the following diagram is commutative:*

$$\begin{array}{ccccc} L & \xleftarrow{\iota_L} & I & \xrightarrow{\iota_R} & R \\ {\scriptstyle m}\downarrow & & \downarrow{\scriptstyle \psi} & & \downarrow{\scriptstyle n} \\ G & \xleftarrow[\eta_L]{} & C & \xrightarrow[\eta_R]{} & H \end{array}$$

Although not necessarily required in this definition, one would often require ι_L, ι_R, m, and n to be injective, that is, they describe embeddings. Even for injective ι_L, ι_R, and m, it is not guaranteed that a valid C and target graph H can be found. For example, if the match m identifies a node in G with a node that is not in the image of ι_L, but this node in G is also the source or target of another edge in G, then the transformation is not possible.

Labeled graphs, as we have defined them in Definition 2.2.1, are a simplification of software models. A more sophisticated formalization is that of typed attributed graphs [37, 90]. In this formalization, nodes and edges have a type, which is defined by a morphism from the abstract syntax graph to a type graph, including some additional structures for inheritance or abstract types. A class in the meta-model corresponds to a node type of node in this typed graph. Similarly, the edge type in the typed graph corresponds to the association relationship in meta-model (type graph).

Furthermore, nodes and edges can have attributes, which are defined mapping nodes and edges into a data signature algebra (Σ-algebra). Both, the type graph and the abstract syntax graph are attributed graphs. The construction of a graph transformation can then be carried over to the category of typed attributed graphs. This construction can be found elsewhere [90].

B.2 Gluing Construction

The double-pushout construction can be seen as a kind of gluing construction. For this, let us focus only on an individual square of the commutative diagram in Definition B.1.2:

$$\begin{array}{ccc} G_1 & \xleftarrow{\phi_1} & I \\ {\scriptstyle i_1}\downarrow & & \downarrow{\scriptstyle \phi_2} \\ G & \xleftarrow[i_2]{} & G_2 \end{array}$$

The *pushout* of G_1 and G_2 along ϕ_1 and ϕ_2 (or short along the *interface* I), is the graph G that is build from the disjoint union of G_1 and G_2 by identifying the nodes and edges with the same pre-image in I:

$$G = (V_1 \cup V_2)/\sim, \quad E = (E_1 \cup E_2)/\sim,$$

where $\sim$ is the equivalence relation that identifies the nodes and edges with the same pre-image in I. This is kind of a gluing of the two graphs G_1 and G_2 along the interface I.

Note that for injective ϕ_1 and ϕ_2, this construction is straightforward, that is one can interpret I as the intersection of G_1 and G_2 and G will then be the union of G_1

and G_2. For non-injective ϕ_1 and ϕ_2, the construction is more involved and there can be splits or merges of nodes.

Applying this gluing interpretation to the double-pushout construction from Definition B.1.2 shows that G can be understood as the result of: 1) gluing L and the context C along I (which defines C in G), and 2) H is the result of gluing R and C along I. That is, one first removes the nodes and edges from G that are not in C and then glues R to C along the interface I.

B.3 Model Transformation Tool Support

The rich theory behind these graph transformations is also the foundation for the tool HENSHIN, which has also utilized in the present thesis. One thing to note here is that the matching is done by HENSHIN using a constraint programming approach which is NP-complete.

Model transformations are described in form of so called HENSHIN rules. These rules consist of a left-hand side (LHS) which describes a pattern to be matched in a graph (i.e., the graph L in the double-pushout construction from Section B.1) and a right-hand side (RHS) which substitutes the LHS (i.e., the graph R in the double-pushout construction from Section B.1). Furthermore, there can be so-called *positive- or negative application conditions* that extend the LHS of the rule. Informally, the application of a HENSHIN rules works as follows:

1. The LHS of the rule (plus possibly positive- or negative application conditions) is matched in the model. There can be many matches (or none). For a concrete edit operation usually one constrains the LHS with parameters (e.g., object ids) to select a concrete match.

2. Nodes and edges in $LHS \setminus RHS$ are removed from the matched part of the model. By "$\setminus$", we mean the set minus operation on $V \cup E$.

3. Nodes and edges in $RHS \setminus LHS$ are "glued" along the preserved elements in $LHS \cap RHS$.

Thanks to the type information the search space for match candidates is usually small in practical examples and therefore the graph transformation approach becomes computationally feasible. Note that there are plenty of other approaches to model transformations, e.g., Fujaba [177] or VIATRA [35]. Furthermore, there are many approaches in the area of exogenous model transformations. More details about this formalization of edit operations can also be found elsewhere [37, 89, 90, 159].

C

A Hypothesis on Paradigm Shifts in Evolution

In Chapter 3 we have focused on a theory of incremental evolution. In this chapter we will develop and sketch a hypothesis about paradigm shifts in the context of the theory of evolution from Chapter 3.

The theory developed in Chapter 3—and also assembly theory—are based on the assumption that the complexity of objects in an ecosystem is increasing. Even if the perceived complexity of objects would align with the *linearly* increasing assembly index, at some point, the complexity of objects would become unmanageable. For example, the edit operations or building blocks of a model would become so complex that it would be impossible to understand them or combine them.

What we can observe in reality is an increase of complexity up to some point in time—then paradigm shifts kind of "reset" the complexity of objects: Occasionally, it happens that a new assembly mechanism for a new "kind" of building blocks comes into existence. The underlying details and mechanisms of earlier paradigms are then hidden inside the new building blocks. Without this *emergence*, it would probably be impossible to handle an ever-increasing complexity of objects.

As an example outside of software, consider "function" of macroscopic rigid bodies in our physical world. For example, for the function of a handle of a cup, only its macroscopic structure is important. Lower level mechanisms, such as how the molecules and atoms are bound together can be neglected by just accepting that there are rigid bodies that we can lift and move without changing their structure. No knowledge about weak or strong physical forces is necessary to understand how to lift a cup of coffee.

Indeed, in the world of software these principles are well-known. Functionality is encapsulated in libraries and an interface is defined to access the functionality. Through the principle of information hiding, the library user is not concerned with the details of its implementation. For example, a user of machine learning frameworks such as TensorFlow or PyTorch is not concerned with the details of the implementation of auto-differentiation. That is, it is only a limited set of building blocks that assembly need to take into account to form new objects from existing ones. By constraining assembly to a subset of already defined objects (defined by a common interfacing mechanism such as a software library ecosystem) an ever-increasing

complexity stays manageable. This also justifies assumptions made earlier that constituents of a pattern subgraph g more likely originated from "recent" assembly indices.

As another example, think about a cloud engineer who builds a standard web application providing some data from a database, even without a single line of code. Under the hood there are software systems, such as the database, that consists of thousands of lines of code. The cloud engineer does not need to handle this complexity at all, but only uses simple interfaces to access the database, while the database itself handles optimization, backups, and other complex tasks. The code of the database runs on a cloud provider's infrastructure, which is again a complex system that the cloud engineer does not need to understand in detail. On an even lower level, electronic circuits are used to run the code, and the cloud engineer does not need to understand the details of the electronic circuits—for example the architecture of the CPU or anything about electronics. That is, the building blocks of the cloud engineer are the interfaces of the cloud provider, the database, and the web application, and the cloud engineer can focus on the interfaces of the cloud provider's building blocks without being distracted by the details of their implementation.

In the realm of software modeling, an example of paradigm shifts would be domain-specific languages. For example, with the emergence of cloud engineering, cloud infrastructure can be described by a domain-specific language. Indeed, abstraction is a key principle of software modeling in general.

An important point to make here is that paradigm shifts are inevitable. The following argument is an extension of an argument by Stuart Kauffman [158]: Suppose we would have a model that describes the evolution of "the entire world". Given a current state, we could use the model to predict a future state. Unfortunately, such a model will likely be impossible to construct. Even if such a model would exist, it will very likely be chaotic and effects such as the butterfly effect will make it impossible to predict the future. We are therefore forced to ignore some details and to make simplifications in a model. Unfortunately, at some point, we will hit the limits of the abstraction, that is, some property that is not captured by the model becomes relevant. For example, let us recall the example of the handle of a cup: If we would only consider the macroscopic structure of the handle, we would not be able to predict that the handle will break if we apply too much force. As Kauffman argues, it is the side effects and the unknowns that often lead to emergence and paradigm shifts. And since we have not every possible effect in our model (otherwise it would be a world model), at some point, we are forced to "break out of the model"—we have to shift to a new paradigm. Indeed, it would be a disadvantage if we integrate things that are irrelevant for the function to be modeled, although, at some point in time, one of the irrelevant properties might become relevant.

We will not study this *shift of paradigms*, that is, when one assembly mechanism is replaced by a new mechanism. Here, we only formulate the following hypothesis and leave its investigation for future work:

Hypothesis

A paradigm—similar to a meta-model—defines what can be constructed and described in a certain domain. New concepts, knowledge, or building blocks emerge from compressing the observation of the domain. A paradigm shift necessitates because of a trade-off that is often made: instead of very expressive paradigms that are typically too "expensive" to be useful in practice, one resorts to less expressive paradigms that are more tailored to the specific needs—and therefore more useful in practice. Concrete triggers for a paradigm shift are:

1. *Hard limitations of the current paradigm are reached, existing problems cannot be solved, or new observations can not be described in the current paradigm. For example, a programming language without any reuse mechanism would be very limited in the complexity of programs that can be written.*

2. Resource constraints *(e.g., work force of engineers and computational resources) limit the complexity of objects that can be constructed, described, or understood in a certain domain. That is,* economic limits *hinder the growth of the domain with the current paradigm.*

3. *A new paradigm automatically emerges on top of the existing paradigm, for example, side effects of objects in the current paradigm become relevant, or new observations can be described in the current paradigm. For example, recently, large language models have emerged to be multitask learners [262], allowing them to be combined using natural language output from one large language model as natural language input of another large language model, leading to new engineering paradigm(s), for example, prompt engineering.*

Indeed, the area of Model-driven Engineering becomes very appealing when considering the hypothesis above. Using meta-modeling the break-out from one model can be modeled itself by utilizing the meta-meta-model. Anyway, we will leave these considerations for future work.

D

Edit Operation Mining

D.1 Ockham Implementation Details

D.1.1 Step 1 – Derivation of Structural Model Differences

Our implementation is based on the Eclipse Modeling Framework. We use the tool SiDiff [162, 284] to compute structural model differences. Our requirements on the model differencing tool are: (1) support for EMF, (2) the option to implement a custom matcher, because modeling tools such as MagicDraw usually provide IDs for every model element, which can be employed by a custom matcher, and (3) an approach to semantically lift model differences based on a set of given edit operations, because we intend to use the semantic lifting approach for the compression of differences in the industrial case study from Chapter 4. Other tools such as EMFCompare could also be used for the computation of model differences and there are no other criteria to favour one over the other. An overview of the different matching techniques is given by [182]; a survey of model comparison approaches is given by [313].

D.1.2 Step 2 – Computation of Simple Change Graphs

We reuse the computation of simple change graphs for model completion. We describe the details of the implementation in Section E.3.1

D.1.3 Step 3 – Frequent Subgraph Mining

Ockham is based on the graph miner Gaston [242], which mines frequent subgraphs by first focusing on frequent paths, then extending to frequent trees, and finally extending the trees to cyclic graphs. Since deciding if a graph G' is a subgraph of another graph G is already $\mathcal{NP}$-complete, there is no polynomial time algorithm for frequent subgraph mining. Usually, if there are large graphs in the database or the threshold is too low, the problem becomes computationally intractable. In these situations, the number of subgraphs typically also becomes intractable. In many

scenarios though, one is not interested in a complete list of all frequent subgraphs. In these cases, huge graphs in the database can be filtered out or the threshold can be increased such that the frequent subgraph mining becomes feasible. We make use of this assumption in our filtering process that is, we assume that a useful edit operation will also appear in small simple change graph components and not only in huge components.

A reasonable support threshold depends on the dataset and the size of the graphs in the dataset. As it would be infeasible to recompute the threshold manually for every dataset, we pre-compute it by running an approximate frequent *subtree miner* for different thresholds up to some fixed size of frequent subtrees. We fix the range of frequent trees and adjust the threshold accordingly. More precisely, we perform a binary search for the threshold until the number of subtrees lies in a predefined range. We use this threshold for the exact subgraph mining. This effectively replaces the support threshold hyper-parameter by a hyper-parameter ST_{max} which satisfies

$$|ST(level = 8, miner)| < ST_{max}$$

where $|ST(level = 8, miner)|$ denotes the number of subtrees found up to level (number of nodes) 8 with the frequent subtree miner used. We therefore do not need to (manually) find a good support threshold for the datasets and, in many cases, we can be sure that the miner is able to terminate in a reasonable amount of time. For frequent subtree mining, we use HOPS [341] because it provides low error rates and good runtime guarantees.

Alternatively, a relative threshold could be used, but we found in a pilot study that our pre-computation works better in terms of average precision.

The choice for GASTON was met in a pilot experiment, where it was superior to other frequent subgraph miners such as GSPAN, and DIMSPAN.

D.1.4 Step 4 – Relevance Reranking

We implement the compression computation and pruning using the *NetworkX*[1] Python library. The algorithm outputs a compression-ranked and pruned list of frequent subgraphs.

Suppose SG is the set of subgraphs obtained from Step 3, we then remove all the graphs in the set

$$SG^{-} = \{g \in SG \mid \exists \tilde{g} \in SG, \text{with } g \subseteq \tilde{g} \\ \wedge \operatorname{supp}(g) = \operatorname{supp}(\tilde{g}) \wedge \operatorname{compr}(g) \leq \operatorname{compr}(\tilde{g})\}.$$

Technically, we do this efficiently by utilizing a subgraph lattice returned from the subgraph miner in Step 3.

1 https://networkx.org/

The output of the relevance reranking is then $SG \setminus SG^{-}$, sorted according to the compression metric (see Definition 6.2.1):

$$\text{compr}(g) = (\text{supp}(g) - 1) \cdot (\mid V_g \mid + \mid E_g \mid)$$

Note there are other graph miners that directly optimize compression measures, most notably SUBDUE [83, 170] and GRAPHMDL [29]. Subdue performs a beam search to discover new structures. In a pilot study, subdue did only discover the ground truth edit operations when we enabled iterative mining and allowed for overlaps. In that case SUBDUE did not terminate on more than 75% of the pilot study datasets. GRAPHMDL [29] was not available when we performed the experiments on edit operation mining and it might be an interesting future work to apply their approach as part of OCKHAM and compare it to the GASTON-based implementation.

D.1.5 Step 5 – Derivation of Edit Operations

We derive HENSHIN transformation rules from the reranked and pruned list of frequent subgraphs in the simple change graphs. The subgraphs of simple change graphs can be directly transferred to HENSHIN transformation rules. A HENSHIN transformation rule consists of a left-hand side graph, and a right-hand side graph, and a matching between elements of the left-hand side graph, and a right-hand side graph (cf. Section B). The added elements are added to the right-hand side graph of the HENSHIN transformation rule, the deleted elements are added to the left hand side graph of the HENSHIN transformation rule, and preserved elements are added to both graphs and a corresponding matching is added to the rule. The meta-model can then be used to complete the edit operations to a consistency preserving edit operations, similar to the derivation of consistency preserving edit operations by Kherer et al. [166]. For example, when a created element requires another element or attribute, which is not yet present, it can be added.

D.2 LLM-based Edit Operation Mining Implementation Details

In this section, we provide details on the implementation of the LLM-based approach for mining edit operations from Chapter 7 We can divide the procedure into two phases: training phase (Step 1, 2, and 3) and the generation phase (i.e., Step 4 and 5).

Training Phase The *input* to the training phase is a set of simple change graphs computed from pair-wise (successive) differences of software models in a model repository. The *output* of the training phase is a fine-tuned language model.

Step (1) – Serialize the graph components: We have to serialize the graph as an edge list. One reason for this is that we want to sample the simple change graphs

edge-wise, as suggested in Section 7.2. Common formats such as the GraphML[2] are not suitable, because they start with a list of vertices before they list the edges. It is intuitive from a language model perspective why this is not a suitable format: Suppose our initial simple change graphs are the results of the application of more than one edit operation. In this case, the initial simple change graphs will be larger than the simple change graphs of the edit patterns that we want to discover. Anyway, the language model will learn to generate serializations statistically similar to the input. This implies that the number of nodes would probably always be too large for an edit operation serialization. We therefore use a simple graph serialization called EdgeList for directed labeled graphs: The serialization of every edge has the format

```
e <src_id> <tgt_id> <edge_label> <src_label> <tgt_label>
```

where <src_id> and <tgt_id> are identifiers for the source and target vertices of the edge, respectively. The serialization of a graph starts with a header line in the format

```
t # <graph_id>
```

and then all edges of the graph serialized line by line.

```
t # 1
e 0 1 Add_port Add_Component Add_Port
e 0 2 Add_requirement Add_Component Add_Requirement
```

Code Listing D.1: An example SCG in the EdgeList format.

An example is given in Listing D.1.

Another degree of freedom that arises in the serialization step is the order of the edges and the identifiers of the vertices in this serialization. After the order of the edges is chosen, we can enumerate the vertices (e.g., increasing integers in the order they appear when following the edges). This still leaves us with $|E|!$ choices (up to automorphism), which can quickly become impractically large. There are *canonical graph serializations* [219], but they do have exponential worst-case time complexity and do not help for our task, because these serializations do not ensure that pattern subgraphs appear with their own canonical ordering[3]. Therefore, choosing *some* possible $|E|!$ edge orderings will probably lead to better results. In this work we chose one ordering (obtained through a depth-first search) that worked well for our data, and we leave the investigation of the influence of further edge orderings for future research.[4]

Step (2) – Randomly split serializations in prompt and completion: As input for the fine-tuning of language model, we provide a set of context and completion pairs. From every EdgeList serialization from the previous step, we generate three training

2 http://graphml.graphdrawing.org/

3 Otherwise graph isomorphism problem and subgraph isomorphism problem would be in the same complexity class, which is not thought to be the case.

4 One of the main reason for this limitation is cost! We used GPT-3 language model series—in particular davinci—in our experiments, which was quite expensive at the time of the experiments.

samples: We compute three cut points. One cut point is randomly chosen in the first 10% of the edges, the second randomly between the first and the last 10% of the edges, and the third cut point randomly among the last 10% of the edges. For every cut point, we then obtain one training sample by taking the edges before the cut point as context and the edges after the cut point as the completion.

Step (3) – Fine-tune a language model: In this step, we use the dataset obtained above to fine-tune a pre-trained language model. Specifically, we use so-called autoregressive language models (the GPT-3 model family), although other types of language models are also conceivable. In autoregressive language models, only the probabilities for the next token in a sequence is predicted based on the context (i.e., the previous tokens). This way, we obtain a language model that is fine-tuned to the simple change graph serializations computed for a specific model repository.

Generation Phase The fine-tuned language model can now be applied to generate edit operations.

Step (4) – Generate simple change graphs for edit operations: For the generation of edit operations, we use an "empty edge" context (i.e., the token `e` to be more precise).

The candidate generation works as follows (see pseudocode in Listing 1): The algorithm takes a set of *incomplete* edit operation candidates (in the form of serialized simple change graphs) and uses the fine-tuned language model to sample new edge candidates and appends them to the incomplete edit operation candidate (Line 12). The sampling generates all possible extensions above a certain probability threshold. Since we cannot guarantee that the extensions lead to a correct EdgeList serialization, we check the syntactical correctness and reject incorrect extensions (Line 13). Furthermore, even syntactically valid extensions could be invalid according to the meta-model and have to be rejected likewise (Line 14). After that, the corresponding simple change graph represents a valid edit operation by definition. Based on a graph isomorphism test, we then filter out duplicates (Line 15). Although graph isomorphism is theoretically expensive from a computational perspective, in our setting, it is acceptable since we have only a few medium size graphs, and employ Weisfeiler-Lehman hashes [137] to speed up the comparison. We add complete candidates to the output list (Line 19) and repeat this process until all candidates are complete (Line 9). Whether a candidate is complete is checked using several conditions such as the total probability of the candidate, a drop in the probability of a generated edge, or a generated stop token.

Step (5) – Ranking of generated candidates: In the last step, we rank the generated candidates from Step 4. Several ranking metrics are conceivable, for example, the probability for a candidate given by the token probabilities of the language model, the compression metric as used in Chapter 6, or also scaled variants of the token probabilities such as scaling with the number of edges or nodes. In the experiments in Chapter 7, we will evaluate different ranking metrics.

E

Model Completion

E.1 Sampling of Experiment Samples

To get a manageable (e.g., we want to perform a manual analysis for correctness and need to keep LLM usage cost acceptable) – but still large enough – sample for our experiments, our aim is to sample around 100–200 examples for each of the datasets (i.e., SYNTHETIC, REPAIRVISION, and INDUSTRY). Every dataset consists of several projects (e.g., submodels in the case of the INDUSTRY dataset), and we ensure that they are represented with the same distribution in our sample. We draw at least 200 examples. Since we ensure that every project is included, at least, once, even if it is very small, this leaves us with 210 samples for the SYNTHETIC dataset, 221 samples for the REPAIRVISION dataset, and 200 samples for the INDUSTRY dataset. For every project in the dataset we used a diversity sampling strategy (cf. Section E.3.3) to obtain a diverse range of samples. For the INDUSTRY dataset, since we perform a manual analysis of semantic correctness there, we further want to reduce the size of the sample, without harming diversity too much. We could have just randomly (or with the procedure above) down-sampled, for example, to 100 samples. Instead, since we generally observe a strong homogeneity in the real-world datasets (i.e., many repeating patterns in INDUSTRY and REPAIRVISION datasets), we wanted to ensure that we do not decrease the heterogeneity. We therefore decided to further select samples from industry with a more controlled procedure:

We further examine the prompt and completion pairs to classify the changes into semantic clusters that we defined. We skimmed the dataset once and came up with a list of change patterns, for example, “interface added between components”. We agreed on a final list of patterns (the first and second author of the paper agreed on categories for the examples from the initially drawn 200 examples). We then recorded whether the training samples contain a change that falls into the same class. We then only included samples that are unique according to the number of training examples in the prompt, the class of the change that we assigned, and whether there is a similar change in the training samples or not (which has also been decided with the help of the patterns we defined for the changes). This leaves 122 samples for the model completion task on the INDUSTRY dataset.

E.2 Approach Formalization

In addition to the terminology from our formalization in Section 2, we will define further operators and then formalize our approach RAMC on top: We define a serialization operator $s\colon \mathcal{G} \to \Sigma^*$, where Σ^* is the set of all strings. This operator s takes a (partial) simple change graph $g \in \mathcal{G}$ and returns a serialization for this graph (the detailed serialization is given in Section E.3.2). Furthermore, we define $\mathrm{r}\colon \Sigma^* \to \Sigma^{*k}$, which, given a (partial) serialized simple change graph $s(g) \in \Sigma^*$, retrieves k "similar" serialized simple change graphs from a (vector) database. We define the prompt operator $\mathrm{prompt}\colon \Sigma^* \times \Sigma^{*k} \to \Sigma^*$, which, given a tuple $(s(g), \mathrm{r}(s(g))$ of the (partial) simple change graph and the retrieved similar serialized simple change graphs, constructs the final prompt (described in detail in Section E.3.4). Given the prompt instruction $i := \mathrm{prompt}((\mathrm{s}(\mathrm{g}), \mathrm{r}(\mathrm{s}(\mathrm{g}))))$, we can generate serialized completed simple change graph candidates by sampling tokens with a LLM. That is, we sample tokens ω_{j+1} from $\mathbb{P}(\omega_{j+1} | i\, \omega_1 \dots \omega_j)$, until the entropy becomes too large or a complete edge serialization has been sampled. A large entropy of the language model token probabilities can be seen as an indicator[1] for a high uncertainty of further tokens. We denote this candidate generation by

$$\begin{aligned} \mathrm{cg}_{\mathrm{LLM}}\colon \Sigma^* &\to \Sigma^* \\ \mathrm{cg}_{\mathrm{LLM}}(i) &\mapsto s(g)\, \omega_1 \dots \omega_J, \end{aligned}$$

where J is the total number of sampled tokens. By $s(g)\, \omega_1$, we denote the string concatenation of $s(g)$ and ω_1, and likewise for the rest of this expression. Finally, we parse $s(g)\, \omega_1 \dots \omega_J$ as a graph (if possible), and interpret it as a completed simple change graph, which represents a model transformation γ. It can happen (although it rarly happens in practise as can be seen in the evaluation section of this paper) that the completed string does not represent our simple change graph serialization format. In this case, we record this failure and consider the model completion as failed. Identifying the parsed graph with the corresponding edit operation, we denote this parsing operator by $s^{-1}\colon \Sigma^* \to \mathcal{E}$. With this notation, the entire model completion approach, RAMC, can be formalized by

$$\begin{aligned} \mathrm{C}_{\mathrm{RAMC}}\colon \mathcal{T} &\to \mathcal{T} \\ (m_1, m_2) &\mapsto \pi(m_1, s^{-1} \circ \mathrm{cg}_{\mathrm{LLM}} \circ \mathrm{prompt} \\ &\qquad \circ\, \mathrm{id} \times \mathrm{r} \circ s \circ \mathrm{SCG}(m_1, m_2)), c \end{aligned}$$

where π is the application operator and SCG the simple change graph operator defined in Section 2.

1 assuming a well-calibrated LLM

E.3 Implementation Details

In this section, we discuss implemention details of the model completion approach. That is, we describe implementation aspects of the operators defined in Section E.2. Note that several of these operators can also be reused in other contexts (e.g., frequent subgraph mining) as well. For example, the computation of simple change graphs—that is, operator SCG—or the serialization —that is, operator *s*—are also used for edit operation mining.

E.3.1 Computation of simple change graphs and their labels

In this section, we describe the realization of the operator SCG, that is, we describe the derivation of the simple change graphs, including their label representations.

In Definition 2.4.4, we defined simple change graphs as subgraphs of a difference graph, which is a labeled graph. In this section, we explain in more detail, how we derive the labels from the models and the change graph, that is, the realization of the operator SCG.

We assume a simplified meta-model, in which we have classes that carry a name, that is, the type of a model element. A class has attributes that have a attribute name and attribute value, and references that have a reference type.

For a given model, we then use this simplified meta-model to derive a labeled graph (cf. Definition 2.2.1): we map objects (i.e., instances of a class) to a node of the labeled graph and instances of references to edges. By this, we ensure that our graph representation is structurally equivalent to an abstract syntax graph of the model (difference). Nodes and edges in the graph carry a label. For nodes, this label is a JSON representation of the object. It has a attribute type with its value equal to the name of the class the object is an instance of. It also contains all attributes with their values for the given object(assuming we can serialize the attribute). More concretely, the attributes are a contained as a nested JSON inside the node label with attribute names equal to the attribute name and JSON value given by the attribute value. Finally, for the edge labels, we use a JSON that has a attribute type with value equal to the reference type.

Next, for the difference graph, we simply add to each node and each edge another JSON attribute changeType, with value equal to Add, Preserve, or Remove, depending on the change type in the difference graph. For modified attributes, we add another node attached to the necessarily preserved object with a JSON label indicating the attribute value *before* and *after* the change. Since a simple change graph is a subgraph of the difference graph, this construction also defines the labels of the simple change graph.

Note that in some cases (e.g., to check for type correctness), we can simply remove attribute information from our labels, thus obtaining a graph that has only

information about the type structure. We use this graph, for example, to check for type correctness of model completions. Furthermore, this graph without attribute information can also be helpful for other use cases, where we are only interested in the type structure, for example, in change pattern mining use cases, or if we want to define a reusable template for edit operations.

Now that we know how to construct a labeled graph for a given model difference, we will next see how we serialize these labeled graphs.

E.3.2 Realization of the serialization

In this section, we describe the realization of the operator s, that is, the serialization format. In this section, we explain our serialization format for graphs, called EdgeList, which will be part of the prompt being send to the LLM. In the language of Section E.2, this section it about the implementation of the operator s.

The serialization of a graph starts with a header line (indicating an id of the graph).

```
t # <graph_id>
```

After the header, all edges of the graph are serialized edge-by-edge, where one edge will correspond to one line in the serialization format. An edge is represented by one line of the following format:

```
e <src_id> <tgt_id> <src_label> <tgt_label> <edge_label>
```

Here, <src_label>, <tgt_label>, and <edge_label> are the labels of the labeled graph corresponding to the simple change graph (cf. Section E.3.1), and <src_id> and <tgt_id> are identifiers for the source and target vertices of the edge, respectively.

An extract of an example simple change graph serialization is given in Listing E.1.

When we designed this serialization format, we had already the application of LLMs for model completion in mind. More common graph serialization formats start with a list of nodes and then list edges between these nodes. Instead, we define nodes implicitly, while defining edges. Therefore, node labels of already defined nodes will be duplicated in our approach. In practise, we avoid this though, by replacing an already defined node label by an *empty JSON*.

Especially in the case of fine-tuning, we do want to avoid that the LLM has first to *guess* the right nodes of the graph before it continues with the edges. The EdgeList format allows for a continuous generation of edges and avoids the break between listing nodes and listing edges.

```
t # 1
e 0 1 {..."add", "type":"port"} {..."add", "type":"component"} {..."add", "type":"port
↪ "}
e 0 2 {..."add", "type":"requirement"} {..."add", "type":"component"} {..."add", "type
↪ ":"requirement"}
```

Code Listing E.1: An example SCG in the EdgeList format.

In a textual representation, we have to linearize also the listing of the edges, that is, we need to decide on an ordering of the edges of the graph. In our case, the order of edges for this serialization is determined using a depth-first search, since it proved to perform best in a pilot study. Nevertheless, other serialization strategies (or even representing the graph in more than one edge order) are conceivable and could be investigated as part of future work.

E.3.3 Realization of the retrieval operator and diversity retrieval

In this section we describe the realization of the operator r, that is, the retrieval of samples for the retrieval augmented generation. RAMC involves retrieving similar examples to the software model the user is currently working on. This retrieval is given by the operator r in Section E.2. To ensure diversity, typical implementations of maximum marginal relevance retrieve elements, element by element, and ensures maximal distance to the already existing elements. This can lead to below optimal samples, because samples that have already been retrieved are later not removed. In essence, typical maximum marginal relevance implementation can get stuck in local optima.

In our sampling algorithm, the goal is the same as in maximum marginal relevance. That is, we want to select samples that are similar to a given input but the samples themselves are diverse. We extend on maximum marginal relevance by using the following retrieval procedure: First, for a given embedding, we retrieve a given number n of elements that are similar to this given embedding. We call this set S. Second, from S, we want to draw another sample of a given size k, that maximizes the distances between all elements. Initially, we draw k random elements from S. Let's call this set D Third, we choose one of these elements e and replace it by an element from $(S \setminus D) \cup \{e\}$ that has maximum distance to the $D \setminus \{e\}$. Finally, we iterate this procedure for a given number of iterations and try to choose at least one element of the initial set D once.

E.3.4 Realization of the candidate generation

In this section, we describe the realization of the operator $\mathrm{cg}_{\mathrm{LLM}} \circ \mathrm{prompt}$, that is, we describe the implementation of the candidate generation. We utilize two different tactics/algorithms to generate candidates for the software model completion. In the language from Section E.2, these represent two different implementations of the operator $\mathrm{cg}_{\mathrm{LLM}} \circ \mathrm{prompt}$. In the first tactic, we keep the control over the sampling procedure and use the language model to generate the completions token-wise. We therefore use this tactic only with a "completion-like" interface. This tactic is more expensive, since we have to process the entire context for every token. Especially for GPT-4, this tactic is not feasible (without major adaptions). For the second tactic, we

utilize the LLM's capabilities to directly generate entire candidate completions. In the present study, we are using this tactic for all completions generated with GPT-4.

Beam-like Sampling Algorithm

The candidate generation works as follows (see pseudo code in Listing 1): The algorithm takes a set of *incomplete* edit operation candidates (in the form of serialized simple change graphs) and uses the (fine-tuned) language model to sample new edge candidates and appends them to the incomplete edit operation candidate (Line 12). The sampling generates all possible extensions above a certain probability threshold. Since we cannot guarantee that the extensions lead to a correct EdgeList serialization, we check the syntactical correctness and reject incorrect extensions (Line 13). Furthermore, even syntactically valid extensions could be invalid according to the meta-model and have to be rejected likewise (Line 14). After that, the corresponding simple change graph represents a valid edit operation by definition. Based on a graph isomorphism test, we then filter out duplicates (Line 15). Although graph isomorphism is theoretically expensive from a computational perspective, in our setting, it is acceptable since we have only a few medium size graphs, and employ Weisfeiler-Lehman hashes [137] to speed up the comparison. We add complete candidates to the output list (Line 19) and repeat this process until all candidates are complete (Line 9). Whether a candidate is complete is checked using several conditions such as the total probability of the candidate, a drop in the probability of a generated edge, or a generated stop token.

ChatModel Instruction

An alternative to the token-wise beam search above is to let the LLM decide when to stop. If multiple candidates should be generated, one could sample with a certain temperature > 0.

For our completion generation, we use the following instruction prompts:

Code Listing E.2: Single edge completion prompt.

```
You are an assistant that is given a list of change graphs in an edge format. That is,
 the graph is given edge by edge. The graphs are directed, labeled graphs. An edge is
serialized as
"e src_id tgt_id edge_label src_label tgt_label"

Labels are dictionaries. If a node appears in more than one edge, the second time it
appears it is replaced by "_" to avoid repetition.

E.g.:
e 0 1 a b bar
e 1 2 bla _ foo

The second edge here would be equivalent to:
"e 1 2 bla bar foo"

There are some change graphs given as examples. Graphs are separated by "\n\n$$\n---\n
".

The last graph in this list of graphs is not yet complete. Exactly one edge is missing
.
```

Algorithm 1 A pattern candidate generation using large language models.

Function GENERATECANDIDATES(ε, $\mathcal{L}$, $\mathcal{TM}$)
Input: ε – given context serialization
    $\mathcal{L}$ – fine-tuned language model
    $\mathcal{TM}$ – metamodel
Output: $[\varepsilon_1, \ldots, \varepsilon_n]$ – list of candidates

incomplete $\leftarrow [\varepsilon]$ {set of incomplete edit operations}
complete $\leftarrow []$ {set of complete edit operations}
while *size*(*incomplete*) > 0 **do**
  ext $\leftarrow []$ {set of extended edit operations}
  for all *op* $\in$ *incomplete* **do**
    ext $\leftarrow$ *ext* + SAMPLEEDGES($\mathcal{L}$, *op*)
    ext $\leftarrow$ CHECKCORRECTSCG(*ext*)
    ext $\leftarrow$ CHECKMETAMODEL($\mathcal{TM}$, *ext*)
    ext $\leftarrow$ PRUNE(*ext*, *complete*)
    incomplete $\leftarrow []$
    for all $\tilde{\varepsilon} \in$ *ext* **do**
      if COMPLETE($\tilde{\varepsilon}$) **then**
        complete $\leftarrow$ *complete* + $\tilde{\varepsilon}$
      else
        incomplete $\leftarrow$ *incomplete* + $\tilde{\varepsilon}$
      end if
    end for
  end for
end while
return *complete*

```
Your task is it, to complete the last graph by guessing the last edge. You can guess
this typically by looking at the examples and trying to deduce the patterns in the
examples. Give this missing edge in the format
"e src_id tgt_id edge_label src_label tgt_label". Note that the beginning "e" is
already part of the prompt.
```

Code Listing E.3: Multiple edge completion prompt.

```
You are an assistant that is given a list of change graphs in an edge format. That is,
 the graph is given edge by edge. The graphs are directed, labeled graphs. An edge is
serialized as
"e src_id tgt_id edge_label src_label tgt_label"

Labels are dictionaries or concatenations of change type and node/edge type. If a node
 appears in more than one edge, the second time it appears it can be replaced by "_"
to avoid repetition.

E.g.:
e 0 1 a b bar
e 1 2 bla _ foo

The second edge here would be equivalent to:
"e 1 2 bla bar foo"

There are some change graphs given as examples. Graphs are separated by "\n\n$$\n---\n
".

The last graph in this list of graphs is not yet complete. Some edges are missing.
Your task is it, to complete the last graph by guessing the missing edges. You can
guess this typically by looking at the examples and trying to deduce the patterns in
the examples. Give the missing edges in the format
"e src_id tgt_id edge_label src_label tgt_label". Note that the beginning "e" is
already part of the prompt. After the last edge of the change graph, add two new lines
.
```

E.3.5 Realization of the projection and simple change graph computation: Evaluation based on graphs

In this section, we describe the realization of the operators π and SCG and how we use this construction to evaluate the model completion in the space of graphs G, instead of based on serializations.

Applying an edit operation to a model m_1 requires some pre-conditions to be fulfilled. This includes the definition of a matching (i.e., to define the preserved model elements). As argued in Section 2, we assume models to be valid and application conditions (pre-conditions) of edit operations to be fulfilled. This is the responsibility of the modeling tool. For example, in the Eclipse Modeling Framework Ecosystem, there are model transformation tools such as HENSHIN [19] that can be used for the definition, verification, and application of an edit operation.

Given a matching and the original (left-hand side) model m_1 the operator SCG and π are inverse to each other (i.e., SCG $\circ\, \pi \equiv$ id under the identification of $\mathcal{E}$ with the range of SCG in $\mathcal{G}$.

The matching defined by the matching of the incomplete edit operation ε can be reused for the matching of the completed edit operation $\gamma \circ \varepsilon$. In case the completed edit operation does not depend on preserved model elements that are not present in the incomplete edit operation, the model completion will be already fully defined by a surrogate operation $C_{\mathcal{G}} : \mathcal{G} \to \mathcal{G}$. In mathematical terms, in this case the following diagram commutes.

$$\begin{array}{ccccc} \mathcal{T} & \xrightarrow{\text{SCG}} & \mathcal{E} \subset \mathcal{G} & \xrightarrow{s} & \Sigma^* \\ {\scriptstyle C_{\text{RAMC}}}\downarrow & & \vdots\,{\scriptstyle C_{\mathcal{G}}} & & \downarrow{\scriptstyle \text{cg}_{\text{LLM}} \circ \text{prompt} \circ \text{id} \times r} \\ \mathcal{T} & \xleftarrow[\pi(m_1,\cdot)]{} & \mathcal{E} & \xrightarrow[s]{} & \Sigma^* \end{array}$$

We can therefore base our evaluation on the surrogate operator $C_{\mathcal{G}}$. For the evaluation, we can therefore sidestep the question of defining π.

Anyway, π can be defined by realized by adding model elements corresponding to "added" nodes to the model, removing model elements corresponding to "removed" nodes from the model, changing attributes of "preserved" nodes given via the change nodes of the simple change graphs, and finally connecting the new model elements to the preserved nodes according to the edges in the simple change graph (and removing dangling edges after removing likewise). Alternatively, tools such as HENSHIN can be used to define an edit rule corresponding to the simple change graph first and then applying this edit rule as an edit operation using the matching defined by the incomplete edit operation ε.

In our implementation and evaluation, we realize the operator SCG as follows: For two (successive) model m_1 and m_2 (i.e., $(m_1, m_2) \in \mathcal{T}$), we compute the simple change graphs in $\mathcal{G}$ as described in Section 2. That is, we first compute a model difference by the tool SIDIFF [284] (other model diffing tools, e.g., EMFCOMPARE, are conceivable). We then map added, removed, and changed model elements to their corresponding nodes in a simple change graph. We then remove all nodes from the matching tree not directly connected to these added, removed, or changed nodes.

Remark E.3.1
Only depending on the matching defined by the incomplete edit operation brings some limitation to the approach. During the software model completion, we can not depend on "new" preserved nodes. Anyway, it is hardly possible, to include the entire model in the context, one therefore needs to take decisions to only bring in slices of the model into the model completions. Alternative slicing options or extensions of the simple change graphs used in this work are conceivable. For example, one could try to extend the simple change graph with preserved nodes that are likely be involved in a subsequent model completion (e.g., model elements that are textually or semantically similar to the elements involved in the current change context). Consideration of these alternatives are beyond the scope of the present work and left for future work.

E.4 Details for Baseline Comparison

In this section, we describe the details of the comparison of our retrieval augmented generation approach to an approach by Chaaben et al. [58].

E.4.1 Evaluated Datasets

A proper level to compare different model completion approaches would be a meta-meta level (following the MOF). This allows us to compare Chaaben et al.'s approach, which works for a subset of UML class diagrams and a subset of activity diagrams, against our approach, which works directly on the abstract syntax graph of the models. By interpreting the abstract syntax of our models as class diagrams, we were able to compare (part of) their approach to ours. In our dataset, we had already classified the changes and the changes that are of interest for the recommendation of new concepts, corresponded to a class of changes we called "Add_node" in our samples. We selected these changes, which left us with 51 samples for the revision dataset.

E.4.2 Evaluation Method

In our comparison, we focused on concept recommendation and association recommendation. Attribute recommendation in class diagrams is quite comparable to concept recommendation. Indeed, in ECORE deciding between a reference to another concept or an attribute of an EClass is more like a design choice and both are considered to be a EStructuralFeature.

We mapped the examples we observed to corresponding concept recommendations. E.g., when an EClass with name User is added via a containment to an EClass with the name Software, we used the tuple [EClass.Software, EClass.User] in Chaaben et al.'s approach. Depeding on the concrete ECore concept, we replaced *name* by the corresponding identifier (e.g., name for EClass, key for EAnnotation, etc.).

To get a clearer picture of the pros and cons of the approaches, we decided to report independently the accuracy of the correct concepts being recommended, the accuracy of correct association being recommended, and we further split in correct type (e.g., EClass in the example above) and correct name (e.g., Software, or User in the example above).

E.4.3 Replication of The Approach

To use the approach introduced by Chaaben et al. [58], as the BASELINE we had to make some design decisions, to make a fair comparison possible.

We utilized the same few-shot examples, following the premise that these could originate from unrelated models. We could have decided to choose the few-shot samples from the dataset, similar to our approach. Anyway, since we consider this a crucial difference of the approach, we used the few-shot samples exactly as in the implementation of their approach. The few-shot samples were loaded from a file that we used in the re-implementation of their approach. We build on their serialization of concepts and enhance the queries by additionally incorporating the partial domain model in a similar manner. We select between one to four pairs of related concepts, enclosing the concept names in brackets.

As in the original approach, we query GPT-3 (text-davinci-002) multiple times to suggest the most frequently occurring concepts. We use the same temperature setting of 0.7 and a maximum token length of 20 tokens, which is sufficient in our REVISION dataset to suggest at least one new pair of concepts. Excess tokens were removed. We also considered upgrading the model used for the BASELINE to GPT-4. This transition would necessitate additional modifications in their approach. When utilizing GPT-4 with the existing prompts, the model begins generating natural language text that is not directly related to the specific use case. This deviation occurs because the text-davinci-002 model is not designed for chat-like interactions, unlike ChatGPT and GPT-4. Consequently, changing the model would require a redesign of their prompts to align with the capabilities of these models.

The authors employed a sampling strategy coupled with a ranking method, so we similarly query GPT-3 multiple times using a variety of prompts, each consisting of the same few-shot examples but with different queries that incorporate a subsets of model elements from the partial model. In their implementation, the authors sampled (random or all) pairs of concepts from the model at hand. This method does not facilitate effective real-time responses, particularly for larger software models, thus making it impractical for scenarios like those encountered in our REVISION dataset (about 685 queries on average, see 5.1). Since we had simple change graphs at hand, we decided to sample the edges/associations from these simple change graphs, ensuring that the number of elements in the partial model for each query ranges between one and four elements.

Furthermore, when suggesting new concepts, the paper considers both elements of each pair as new concepts. Unfortunately, one of these elements is usually already present in the partial model. This results in existing model elements being ranked at the top, rather than new concepts, leading to poorer outcomes. We have improved upon this by filtering out classes that already exist in the model.

It is noteworthy that several recommender systems make a list of k recommendations. We did intentionally decide to set k equal to one, since in a real world

scenario (compare to GitHub Copilot), one would typically not come up with a list, but directly integrate one recommendation in the IDE. To ensure a fair comparison, we focus on the single most frequently occurring completion.

E.5 Few-shot Examples

We present the concrete few-shot samples belonging to the running example from Section 8.2. These examples demonstrate the retrieval of, in this specific case, four few-shot instances through our vector store. The similarity-based retrieval mechanism is further detailed in Section E.3.3.

```
t # 5175
e 2 1 "{'changeType': 'Add', 'type': 'reference', 'referenceTypeName': 'eOperations'}"
 "{'changeType': 'Preserve', 'type': 'object', 'className': 'EClass', 'attributes': {'
id': '_ftfz6d6tEei97MD7GK1RmA', 'eAnnotations':['org.eclipse.emf.ecore.impl.
EAnnotationImpl@1d8d14f1 (source: http://www.eclipse.org/emf/2002/GenModel)','org.
eclipse.emf.ecore.impl.EAnnotationImpl@c8ca1dd (source: duplicates)'],'name':'
Classifier','ePackage':'uml','abstract':'true','interface':'false', 'eIDAttribute':'
name','eStructuralFeatures':['isAbstract','generalization','powertypeExtent','feature
','inheritedMember','redefinedClassifier','general','substitution','attribute','
representation','collaborationUse','ownedUseCase','useCase'],'eGenericSuperTypes':['
org.eclipse.emf.ecore.impl.EGenericTypeImpl@239c2926 (expression: Namespace)','org.
eclipse.emf.ecore.impl.EGenericTypeImpl@526bc7ba (expression: RedefinableElement)','
org.eclipse.emf.ecore.impl.EGenericTypeImpl@6999e7c8 (expression: Type)','...']}}" "{'
changeType': 'Add', 'type': 'object', 'className': 'EOperation', 'attributes': {'id':
'_mrycqN6tEei97MD7GK1RmA', 'name':'getAllUsedInterfaces','ordered':'false','unique':'
true','lowerBound':'0','upperBound':'-1','many':'true','required':'false','eType':'
Interface','eGenericType':'org.eclipse.emf.ecore.impl.EGenericTypeImpl@762545f6 (
expression: Interface)','eContainingClass':'Classifier'}}"
e 2 0 "{'changeType': 'Add', 'type': 'reference', 'referenceTypeName': 'eOperations'}"
 _ "{'changeType': 'Add', 'type': 'object', 'className': 'EOperation', 'attributes':
{'id': '_mrycp96tEei97MD7GK1RmA', 'name':'getUsedInterfaces','ordered':'false','unique
':'true','lowerBound':'0','upperBound':'-1','many':'true','required':'false','eType':'
Interface','eGenericType':'org.eclipse.emf.ecore.impl.EGenericTypeImpl@3d23f56e (
expression: Interface)','eContainingClass':'Classifier'}}"

$$
---
t # 1250
e 2 1 "{'changeType': 'Add', 'type': 'reference', 'referenceTypeName': 'eOperations'}"
 "{'changeType': 'Preserve', 'type': 'object', 'className': 'EClass', 'attributes': {'
id': '_ftfz6d6tEei97MD7GK1RmA', 'eAnnotations':['org.eclipse.emf.ecore.impl.
EAnnotationImpl@50bd114f (source: http://www.eclipse.org/emf/2002/GenModel)','org.
eclipse.emf.ecore.impl.EAnnotationImpl@11c9b440 (source: duplicates)'],'name':'
Classifier','ePackage':'uml','abstract':'true','interface':'false','eIDAttribute':'
name','eStructuralFeatures':['isAbstract','generalization','powertypeExtent','feature
','inheritedMember','redefinedClassifier','general','ownedUseCase','useCase','
substitution','attribute','representation','collaborationUse','ownedSignature'],'
eGenericSuperTypes':['org.eclipse.emf.ecore.impl.EGenericTypeImpl@1504a6f7 (expression
: Namespace)','org.eclipse.emf.ecore.impl.EGenericTypeImpl@65db7f4d (expression:
RedefinableElement)','org.eclipse.emf.ecore.impl.EGenericTypeImpl@225a383c (expression
: Type)','...']}}" "{'changeType': 'Add', 'type': 'object', 'className': 'EOperation',
 'attributes': {'id': '_inuJYt6tEei97MD7GK1RmA', 'name':'getOperation','ordered':'
false','unique':'true','lowerBound':'0','upperBound':'1','many':'false','required':'
false','eType':'Operation','eGenericType':'org.eclipse.emf.ecore.impl.
EGenericTypeImpl@4e5c0171 (expression: Operation)','eContainingClass':'Classifier','
eParameters':['name']}}"
```

```
e 1 0 "{'changeType': 'Add', 'type': 'reference', 'referenceTypeName': 'eParameters'}"
 _ "{'changeType': 'Add', 'type': 'object', 'className': 'EParameter', 'attributes':
{'id': '_inuJY96tEei97MD7GK1RmA', 'name':'name','ordered':'false','unique':'true','
lowerBound':'1','upperBound':'1','many':'false','required':'true','eType':'String','
eGenericType':'org.eclipse.emf.ecore.impl.EGenericTypeImpl@bbcf831 (expression: String
)','eOperation':'getOperation'}}"

$$
---
t # 2292
e 0 2 "{'changeType': 'Remove', 'type': 'reference', 'referenceTypeName': '
eAnnotations'}" "{'changeType': 'Preserve', 'type': 'object', 'className': 'EClass', '
attributes': {'id': '_fthA796tEei97MD7GK1RmA', 'eAnnotations':['org.eclipse.emf.ecore.
impl.EAnnotationImpl@2fa33653 (source: http://www.eclipse.org/emf/2002/GenModel)','org
.eclipse.emf.ecore.impl.EAnnotationImpl@59d423ca (source: duplicates)'],'name':'
Extension','ePackage':'uml','abstract':'false','interface':'false','eOperations':['non
_owned_end','is_binary','getStereotype','getStereotypeEnd','isRequired','getMetaclass
','metaclassEnd'],'eStructuralFeatures':['isRequired','metaclass'],'eGenericSuperTypes
':['org.eclipse.emf.ecore.impl.EGenericTypeImpl@3ff99636 (expression: Association)
']}}" "{'changeType': 'Remove', 'type': 'object', 'className': 'EAnnotation', '
attributes': {'id': '_oBpkOd6tEei97MD7GK1RmA', 'source':'http://www.eclipse.org/emf
/2002/GenModel','details':['org.eclipse.emf.ecore.impl.EStringToStringMapEntryImpl@95f
12e0 (key: documentation, value: An extension is used to indicate that the properties
of a metaclass are extended through a stereotype, and gives the ability to flexibly
add (and later remove) stereotypes to classes.)'],'eModelElement':'Extension'}}"
e 2 3 "{'changeType': 'Remove', 'type': 'reference', 'referenceTypeName': 'details'}"
_ "{'changeType': 'Remove', 'type': 'object', 'className': 'EStringToStringMapEntry',
'attributes': {'id': '_oBpkOt6tEei97MD7GK1RmA', 'key':'documentation','value':'An
extension is used to indicate that the properties of a metaclass are extended through
a stereotype, and gives the ability to flexibly add (and later remove) stereotypes to
classes.'}}"
e 0 4 "{'changeType': 'Add', 'type': 'reference', 'referenceTypeName': 'eAnnotations
'}" _ "{'changeType': 'Add', 'type': 'object', 'className': 'EAnnotation', 'attributes
': {'id': '_0oByC96tEei97MD7GK1RmA', 'source':'http://www.eclipse.org/emf/2002/
GenModel','details':['org.eclipse.emf.ecore.impl.EStringToStringMapEntryImpl@5cd02377
(key: documentation, value: An extension is used to indicate that the properties of a
metaclass are extended through a stereotype, and gives the ability to flexibly add (
and later remove) stereotypes to classes.\\n<p>Merged from package UML (URI {@literal
http://www.omg.org/spec/UML/20110701}).</p>)'],'eModelElement':'Extension'}}"
e 4 1 "{'changeType': 'Add', 'type': 'reference', 'referenceTypeName': 'details'}" _
"{'changeType': 'Add', 'type': 'object', 'className': 'EStringToStringMapEntry', '
attributes': {'id': '_0oByDN6tEei97MD7GK1RmA', 'key':'documentation','value':'An
extension is used to indicate that the properties of a metaclass are extended through
a stereotype, and gives the ability to flexibly add (and later remove) stereotypes to
classes.\\n<p>Merged from p'}}"

$$
---
t # 88
e 0 2 "{'changeType': 'Add', 'type': 'reference', 'referenceTypeName': 'eAnnotations
'}" "{'changeType': 'Preserve', 'type': 'object', 'className': 'EClass', 'attributes':
 {'id': '_fZD13N6tEei97MD7GK1RmA', 'eAnnotations':['org.eclipse.emf.ecore.impl.
EAnnotationImpl@68481491 (source: http://www.eclipse.org/emf/2002/GenModel)','org.
eclipse.emf.ecore.impl.EAnnotationImpl@4faef368 (source: duplicates)'],'name':'
DataType','ePackage':'cmof','abstract':'false','interface':'false','eIDAttribute':'
name','eStructuralFeatures':['ownedOperation','ownedAttribute'],'eGenericSuperTypes
':['org.eclipse.emf.ecore.impl.EGenericTypeImpl@7ebe7d9f (expression: Classifier)']}}"
 "{'changeType': 'Add', 'type': 'object', 'className': 'EAnnotation', 'attributes': {'
id': '_ffDLSt6tEei97MD7GK1RmA', 'source':'http://www.eclipse.org/emf/2002/GenModel','
details':['org.eclipse.emf.ecore.impl.EStringToStringMapEntryImpl@5e553e0a (key:
documentation, value: A data type is a type whose instances are identified only by
their value. A data type may contain attributes to support the modeling of structured
data types.)'],'eModelElement':'DataType'}}"
```

```
e 2 1 "{'changeType': 'Add', 'type': 'reference', 'referenceTypeName': 'details'}" _
"{'changeType': 'Add', 'type': 'object', 'className': 'EStringToStringMapEntry', '
attributes': {'id': '_ffDLS96tEei97MD7GK1RmA', 'key':'documentation','value':'A data
type is a type whose instances are identified only by their value. A data type may
contain attributes to support the modeling of structured data types.'}}"
e 3 4 "{'changeType': 'Remove', 'type': 'reference', 'referenceTypeName': 'details'}"
"{'changeType': 'Remove', 'type': 'object', 'className': 'EAnnotation', 'attributes':
{'id': '_fZD13d6tEei97MD7GK1RmA', 'source':'http://www.eclipse.org/emf/2002/GenModel
','details':['org.eclipse.emf.ecore.impl.EStringToStringMapEntryImpl@77d8e24f (key:
documentation, value: A data type is a type whose instances are identified only by
their value. A DataType may contain attributes to support the modeling of structured
data types.\\n\\n\\n\\nA typical use of data types would be to represent programming
language primitive types or CORBA basic types. For example, integer and string types
are often treated as data types.\\r\\nDataType is an abstract class that acts as a
common superclass for different kinds of data types.)'],'eModelElement':'DataType'}}"
"{'changeType': 'Remove', 'type': 'object', 'className': 'EStringToStringMapEntry', '
attributes': {'id': '_fZD13t6tEei97MD7GK1RmA', 'key':'documentation','value':'A data
type is a type whose instances are identified only by their value. A DataType may
contain attributes to support the modeling of structured data types.\\n\\n\\n\\nA
typical use of data types would be to'}}"
e 0 3 "{'changeType': 'Remove', 'type': 'reference', 'referenceTypeName': '
eAnnotations'}" _ _

$$
```

E.6 Further Preprocessing and Filtering Steps

We perform some additional filtering steps during the (pre-)processing of simple change graphs and the sampling. For the sake of clarity, we omitted them in the description of the approach and experiment description. The applied filters are the following:

- Because we have a limited context size available for the large language models, very long attribute descriptions (for example in comments) are limited to a length of 200 characters. Everything longer than 200 characters has been cut and "…" are appended.
- When sampling few-shot samples, and the overall prompt size becomes too long, we remove few-shot samples until the prompt fits into the model.
- Serialized simple change graphs that are too large to fit in the context of the language model are filtered.
- To save tokens and therefore reduce language model usage costs, we do not repeat node labels, but instead replace them by a "_" token (or "{}" when using JSON in the label representation).
- We filtered duplicated simple change graphs.

- Models from the original REPAIRVISION dataset that could not be loaded or had empty history were removed. The description of the dataset parameters in Section 8.3.2 describes the state after this filtering.

E.7 Detailed Results of the Industry Dataset and Experiment 3

Table E.1 summarizes the results of our results from Experiment 3 . We distinguished between four completion task characteristics in the rows of the table: noise present, project specific change, complex change, and reoccurring pattern. All four are binary relations. "Noise Present" indicates whether there are changes entangled in the task or in the few-shot examples. We considered the task to be a "Project Specific Change", if the change was not common for the modeling language (SysML), but rather some pattern we observed for this project only. "Complex Change" indicates a change that does consist of several interconnected atomic changes (e.g., adding an attribute or adding a class, i.e., implicitly we distinguish between atomic changes and complex changes, as common in the field [166]). Finally "Reoccurring Pattern" describes if we observe the pattern of the task at hand also in the few-shot samples. That is, aside from concrete attribute values, the change happens often in the project and can be retrieved via our semantic retrieval. Correctness is classified as follows in the columns of the table: We only consider semantically correct completions (evaluated via a manual analysis) as *correct*. The *incorrect* completions, we further classify in "semantically conceivable" (i.e., the change is not the one observed in the ground truth, but without further context it would also be meaningful), "Semantically Incorrect" (i.e., format correct, structurally correct, but the completion has a meaning different from the ground truth completion), "structurally incorrect" (i.e., a reference connects no the right nodes, or a new class is associated to another class where nothing should be added, etc.), and "format incorrect" (i.e., without error correction, the graph serialization could not be parsed, e.g., because an existing node id is reused by another node).

Remark E.7.1
In Section 8.3, we stated that there is no significant relationship between the correctness and the number of few-shot examples that are added to the prompt. For the INDUSTRY *dataset, we additionally recorded, if (at least one) similar pattern is among the few-shot examples. Separating these two cases – i.e., there is a similar pattern among the few-shot examples or not – we see that in the first case there is no significant relationship between the number of few-shot examples, while in the second case there is a significant relationship.*

Table E.1: Comparison of different failure types along several characteristics of the completion task. This table summarizes the results of our manual analysis of the INDUSTRY dataset.

Task Characteristic		Level of Correctness					Total	Total (%)
		Correct	Incorrect					
		Semantically Correct	Semantically Conceivable	Semantically Incorrect	Structurally Incorrect	Format Incorrect		
Noise Present	TRUE	30.77%	23.08%	15.38%	15.38%	15.38%	13	11
	FALSE	66.06%	14.68%	8.26%	4.59%	6.42%	109	89
Project Specific Change	TRUE	74.55%	3.64%	7.27%	5.45%	9.09%	55	45
	FALSE	52.24%	25.37%	10.45%	5.97%	5.97%	67	55
Complex Change	TRUE	66.67%	23.81%	4.76%	0.00%	4.76%	21	17
	FALSE	61.39%	13.86%	9.90%	6.93%	7.92%	101	83
Reoccurring Pattern	TRUE	71.13%	17.53%	6.19%	2.06%	3.09%	97	80
	FALSE	28.00%	8.00%	20.00%	20.00%	24.00%	25	20
Total		62.30%	15.57%	9.02%	5.74%	7.38%	122	100

This suggests that as long as a similar few-shot example is available, the amount of few-shot examples does not matter too much, while in the case that the examples are rather unrelated, the amount plays a role.

E.8 Detailed Results of the Fine-Tuning Experiments

This section delves into a detailed analysis of our last experiment highlighting the influence of various factors on the *average token accuracy*. We are especially interested in the model token accuracy of the fine-tuned language model in relationship with the properties of the dataset and the properties of the fine-tuning such as the number of fine-tuning epochs and the base language model used. We fine-tune one LLM per simulated repository. As base models we choose text-ada-001, text-curie-001, and text-davinci-003 from the GPT-3 family. Since fine-tuning the text-davinci-003 model is quite expensive (i.e., 3 Cents per thousand tokens at the time of this experiment), we fine-tuned this model only for the model repositories where the perturbation probability equals 100% (the ones which are typically the harder ones). This leaves us with a total of 112 fine-tuned models (24 simulated repositories for text-ada-001 and text-curie-001 and 8 for text-davinci-003, which is 24*2*2+ 8*2*1 = 112) and a total fine-tuning cost of 347US$. Building on the insights previously touched upon, our analysis reveals a strong correlation between average token accuracy and the number of fine-tuning epochs. Furthermore, it becomes evident that larger models exhibit better performance in terms of average token accuracy. Regarding the repository properties, we only find significant negative correlations with the perturbation probability (Table E.2). We therefore also analyze model completions from a graph matching perspective (like already mentioned in Experiment 4). Since generating all completion candidates for all test samples of all fined-tuned language models would be even more expensive, we select two fine-tuned language models, the less cost-intensive alternative, and perform the analysis of the model completions on them.

E.9 Detailed Related Work

Since it constitutes an important challenge in model-based software engineering, there have been several approaches to model completion in the past. Except for simple refactoring or repair operations, it is certainly fair to say that none of the more sophisticated approaches has been adopted by practitioners. Also, as we have discussed in the introduction in Chapter 1, a quantitative comparison of existing

Table E.2: Pearson correlations of the average token accuracy w.r.t. several properties. Repo_D denotes the number of revisions, Repo_E the number of applied edit operations, and Repo_P the perturbation probability.

	Repo_D	Repo_E	Repo_P	Epochs	Token Count	Base Model
Average Token Accuracy	0.16	0.13	-0.22*	0.69**	0.08	0.43**
Token Accuracy (All)	0.16	0.13	-0.22*	0.69**	0.08	0.43**
Token Accuracy (Ada)	0.26	0.22	-0.43*	0.72**	0.14	–
Token Accuracy (Curie)	0.13	0.12	-0.35*	0.82**	0.02	–
Token Accuracy (Davinci)	0.02	-0.04	–	0.94**	-0.06	–

(**: $p < .001$, *: $p < 0.05$)

approaches is quite difficult for several reasons. In this section, we therefore present a detailed discussion of model completion and related approaches.

Current Challenges in the Research Domain

Research in Model-driven Engineering faces several challenges that should receive increased attention in the future.

The scarcity of reusable datasets [45, 75, 209, 268] for many use cases in Model-driven Engineering hinders the comparison of different approaches, which is then often reduced to a qualitative analysis. The lack of proper datasets also poses a challenge for the development and evaluation of data-driven (e.g., machine learning) approaches in Model-driven Engineering. To circumvent this lack of datasets, many authors in Model-driven Engineering research report on experiences using their approaches in a concrete application context, that is, as part of a tool. Reporting on an evaluation in a concrete application setting, again, makes it difficult to compare against the approach, especially if the application context or the tool is not available to the public and/or user studies are performed.

There are only a few datasets available that can be used to evaluate model completion. In the concrete example of model completion, the evaluation is often performed on a dataset of model snapshots, from which elements are removed artificially. Instead, it would be more realistic to have pairs of to-be-completed models and their completed counterparts.

Finally, there are no commonly accepted evaluation metrics and often technologies or proposed approaches are evaluated in a manner that is only applicable for the specific use case at hand. Only a minority of the literature reports on metrics that are independent of their specific approach and only depending on the use case (i.e., model completion). For instance, a model completion methodology could suggest the top-10 names for meta-model classes for inclusion in a meta-model, with the eval-

Table E.3: Related work summary.

Paper	Task	Method	Evaluation	Data	Prerequisites (for evaluation)	History	Comparison Possible?
[5] [312]	Single-step operations and similar, related Simulink systems	Information retrieval (association rule mining, frequency-based matching)	Metrics analysis (prediction, accuracy, error classification)	Simulink (available)	None	No	With adaption: Adaption of RAMC to Simulink datasets beyond the scope.
[129]	Library block recommendation	Information retrieval (association rules, collaborative filtering)	Metrics analysis (precision, recall, and F-measure)	Simulink (available)	None	No	With adaption: Adaption of RAMC to Simulink datasets beyond the scope.
[7] [8]	Recommendation of related classes, possible sub- or super-classes, relationships between elements, element names	Knowledge graphs, semantic web technologies	Planned user study but not yet conducted (no metric)	UML (not available)	Conceptual knowledge bases (semantically related terms, built from natural language data) and semantic network (not available)	No	No: Dataset and (parts of) their approach are not available.
[79]	Activity node recommendation	Information retrieval (similarity-based, pattern mining, pattern as relationships between activity nodes)	Metrics analysis (HitRate, Precision, Recall, and F1 Score)	Business process modeling (not available)	Database constructed from existing processes (not available)	No	No: Their dataset is not available to us and the approach is domain specific (BPMN).
[80]	Recommendation of entities in metamodels (classes, structural features), no support for types of the recommended attributes, relationships	Information retrieval (similarity-based, collaborative filtering strategy)	Metrics analysis (rather best case scenario – out of N items some of the (possibly huge) model are correct – (success rate (SR@N), precision, recall, F1 score))	Ecore metamodels (available)	Predefined categories/labels beneficial	No	With adaption: Adaption of their approach to our history-based data possible but beyond the scope of this work. Note: Only sub-tasks of model completion.
[92]	Recommendation of new concepts (i.e., class names), attributes, operations	Clustering algorithm (on semantic relations)	User study (relevant and new recommendations (PN), non useful recommendations (NU) not recommended but included in individual design (NR), relevant rate (TP), accuracy rate of new suggestions (TN))	UML (not available)	Clustered UML diagrams (not available)	No	No: Use cases are slightly different. They recommend classes and adapt with user feedback, we recommend model elements (in "small steps").
[181]	Model completion	Pattern matching and Association rule mining	Metrics analysis (Precision)	Eclipse GMF Project meta-models (available)	Catalog of change patterns (not available)	Yes	No: We could apply our approach to their data, but their numbers are reported based on the concept of edit rule applications and therefore can not be compared. Anyway, an adaption and reimplementation of their evaluation seems reasonable but is beyond the scope.
[193] [194] [236]	Model completion	Rule-based pattern matching	User study (number of saved user actions, time against manual completion)	UML (not available)	Catalog of change patterns (not available)	Yes	No: Dataset used for evaluation is not available to us and running their approach on our data requires a catalog of change patterns that is not available. As for [181], reimplementation of evaluation seems reasonable but is beyond the scope.
[82]	Recommending new concepts (class names) and attributes	Information retrieval (similarity-based, graph kernels, TF-IDF)	Metrics (success rate, precision, recall, and F-measure (modified)) analysis (best-case scenario, i.e., check whether one out of N recommendations correct – evaluated on "token" level, structural correctness, e.g., a new class connected to >= 2 other classes, not evaluated and not reflected in the approach)	ModelSet, reverse engineered class diagrams from Java code, JSON crawled from GitHub, Ecore metamodels (partially available)	None	No	With adaption: Structural correctness not reflected by their approach. Adapting their approach to compare for concept and attribute recommendation seems reasonable. Note: Only sub-tasks of model completion.
[58]	Recommending new concepts (class names), attributes, association names	Machine Learning (GPT-3, few-shot learning)	Metrics analysis (30 models selected and evaluated manually, precision, recall)	ModelSet (available)	None	No	With adaption: We adapted their approach to work on EMF-based models (interpreting them as class diagrams).
[48]	"Contextualized" model completion	Machine Learning (reuse pre-trained word embedding models, project-specific training, NLP-based system, word embedding similarity based on textual information)	Metric analysis (Precision, Recall) – best case scenario, out of N items	Industrial data (incident management system in municipal water supply and sewage in Malaga)(not available)	Textual information required (of project and/or related business domain)(not available)	No	No: Their dataset not available to us and other artifacts not (explicitly) available in our dataset.
[81]	Recommendation of edit operations given preceding edit operations (on a type level, i.e., omitting attribute values)	Machine Learning (Encoder-Decoder LSTM neural network)	Metric analysis, limited possibilities due to ignoring names and values, out of N items some are correct, unclear how many items get recommended (Success rate, precision recall)	BPMN	None	Yes	No: Their approach depends on operation recording, which we do not have available for our datasets. Regarding their BPMN dataset, we could not find models (+ metamodel) for an application of our approach to their dataset.

uation of this method focusing solely on the accuracy of these ten recommendations. Consequently, this creates a challenge in directly comparing such an approach to others that might recommend a single name while also suggesting relationships between the newly added class and existing classes. Further it would require a ground truth of to-be-completed and completed models, which is not available for most datasets. But even here, its not easy to define what a correct completion is. For example, if a model element is missing in the incomplete model, but the model element is not required for the model to be valid, is it a correct completion or not? Likewise, if a recommended class name is a synonym of the correct class name, is it a correct completion or not? Note that for source code, there are commonly accepted datasets such as HumanEval [62] and evaluation metrics [62] to evaluate code completion approaches. For example, since there is a well-defined execution semantics, the evaluation of a code completion approach can be performed by checking the correctness of the code completion in a test suite. For many models (e.g., UML, SysML, Ecore, etc.), there is no well-defined execution semantics and therefore a test approach for evaluation would not be applicable to software models, in general.

Comparison and differentiation from other approaches

In Table E.3, we summarize the related work with a specific focus on the model completion task. For each approach, we included information about the specific task, the method used and the evaluation process, including the data used for evaluation and specific prerequisites are required for replicating the evaluation. Given the sometimes challenging nature of tracking the availability of artifacts, we acknowledge that some information might not be entirely accurate, and we apologize for any inadvertent inaccuracies. A main finding of our analysis of related work is that, currently, a direct comparison with other approaches, for most approaches, is infeasible, due to the field's novelty, the absence of commonly accepted metrics and datasets, or a focus of previous work on artificial datasets without real-world model evolution. We also highlight the most prevailing reason why a comparison to our approach is infeasible in Table E.3. In the following paragraph, we will delve into the reasons why a direct comparison to the approaches in Table E.3 is not easily possible.

When we want to compare our approach to an existing approach, we could do this comparison on he data the approach was evaluated on. For this, the exact (test) data and a comparable metric need to be published. If this is not the case, it might still be possible to perform a comparison by applying the existing approach to our data. But for this the approach should either be generic (i.e., not depend on a specific domain) or work for our datasets (i.e., EMF models).

(Partial) model completion

The approach by Agt-Rickauer et al. [7, 8] focuses on suggesting related class names, potential sub- or super-class names, and different names for connections given a specific focus point in the model. However, their approach does not extend to

suggesting attributes, operation names, or relationship types, making a comparison to our approach challenging. Conversely, our approach goes far beyond suggesting the names of new elements. Furthermore, the evaluation of their method is deeply integrated within the tool, making a direct comparison impossible. This approach relies on conceptual knowledge bases (comprising semantically related terms built from natural language data) and a semantic network, neither of which are available for external validation. Consequently, it is challenging to verify when and how the suggestions are semantically and structurally correct.

The approach by Elkamel et al. [92] suggests entities in metamodels, such as classes and structural features, but does not support types for the recommended attributes or relationships. In an offline phase, they use a clustering algorithm to partition UML classes collected from various UML class diagrams based on the semantic relations between their characteristics. Subsequently, they recommend semantically similar whole classes, and individual methods and attributes of that class can be accepted or rejected. Their approach is not based on historical data. On the other hand, we cannot apply their approach to our data, as they only suggest entire classes while our approach focuses on a more general setting. Their approach directly relies on user feedback, as entire classes are suggested for the user including its attributes to accept or reject. As a result, it is rare for an entire class to be completely correct initially. While their focus is more on the user setting, we focus on the core effectiveness of the LLM technology. This makes a direct comparison between our methods unintuitive.

Similarly, Di Rocco et al. [80] propose an approach for suggesting new classes and structural features (attributes and references) in metamodels. Their method generates recommendations as a ranked list of classes if the active context is a package, or as a ranked list of structural features if the active context is a class. This approach involves identifying a subset of the most similar metamodels from given metamodel repositories and determining the most similar contexts within that subset. However, the method lacks support for recommending the types of attributes and relationships.

Di Rocco et al. [81] further present a recommender system that uses an Encoder-Decoder neural network to assist modelers with performing editing operations. The system learns from past modeling activities and is evaluated on a BPMN dataset. These past activities are modeled as edit operation sequences. One limitation of this specific format is that the changes of an element in the edit operation sequences can be scattered throughout the complete sequence, with possibly hundreds of other edit operations between them. This also means that connected/related elements or elements that belong together can appear at completely different locations within such a sequence. These connected/related elements can give valuable context to the model completion task. If then, as in the work by Di Rocco et al. [81], only the last 10 edit operations are considered, important information regarding the local (graph-like context) might be lost. This issue becomes more pronounced as models increase in size. One could instead not only focus on the last x edit operations, but instead put the entire history of a model into the LLM context. Anyway, especially

for large models, providing the entire history of the model as context is infeasible and may not fit in the context of an LLM.

We cannot compare their approach to our model completion approach because it does not include the specific details and values of operations. For instance in the example in Listing E.4.

```
set-att name BPMN2ActionContributor to #200
```

Code Listing E.4: An example of the NEMO [81]

In the setAtt operation, the class name being created isn't suggested. Instead, each event is simplified to a tuple <setAtt, class, name>. While their approach focuses on proposing simplified completions, as highlighted in their work, our approach suggests more complex model elements with detailed, specific values (e.g. class names, concrete attribute values). An example of a concrete, linearized model completion suggestion of RAMC is given Listening E.5.

```
1 2 "{'changeType': 'Add', 'type': 'reference',
'referenceTypeName':'eOperations'}" _ "{'changeType':'Add',
'type': 'object', 'className': 'EOperation', 'attributes'
{'id':'_IU7gFt6tEei97MD7GK1RmA','name':
'getMetaclass','ordered':'false','unique':'true',
'lowerBound':'0','upperBound':'1','many':'false',
'required':'false','eType':'Metaclass',
'eGenericType':'Metaclass','eContainingClass':
'Extension'}}"
```

Code Listing E.5: A RAMC completion candidate

The competition candidate includes a specific change to the model, detailing how it is connected to other elements (as a quick reminder, 1 and 2 represent the source and target nodes, followed by the edge attributes in the first). It suggests specific values and the names of changed attribute elements and much more. All in all, comparing our work to the work by Di Rocco et al. [81] would be an unfair comparison for both sides.

Di Rocco et al. [82] focus on model completion by suggesting new classes and structural features (attributes, references, methods, and fields). This is achieved by constructing a separate graph for each class in the models and use graph kernel similarity to identify the most similar items among the training set to the partial model that should be completed. However, their approach does not ensure structural correctness, such as where to add a class or how elements should be connected overall. Additionally, the work reports very low precision and recall values, which left us believing that their method will likely not perform well on real-world and industrial datasets. For our baseline, we therefore decided to reimplement the approach by Chaaben et al. [58], which is also more closely related to our approach.

Regarding the use of language models, Chaaben et al. [58] use LLMs and acknowledge that their results are preliminary, considering only a few UML examples (30 domain models, selected manually from the dataset ModelSet). Their evaluation fo-

cuses on suggesting class names, attributes of classes and association names. They do not consider historical data, therefore without major adaptation, we cannot perform our experiments on their data. We primarily focus on historical data because we aim to gather real-world examples. Instead of randomly excluding model components and using them as ground truth, we study actual real-world evolution, making the setting much more realistic. Additionally, their approach does not scale for larger models, so our real-world models are by far too large to fit within the context window of the GPT-3 model (text-davinci-002). This challenge is precisely why we decided to focus on historical data in combination with simple change graph slicing. Anyway, since the work by Chaaben et al. [58] is the most closely rated work, we re-implement their approach and tailor it to our dataset to enable a direct comparison between their method and ours.

Additional data required for model completion

There are other approaches, such as those employing rule-based matching based on a predefined catalog of change patterns (edit operations), where additional data is required to compare their approach to ours [181, 193, 194, 236]. These pattern-based approaches are to some extend orthogonal to ours. For example, one can use semantic lifting [166], to further compress change graphs before applying an approach like ours. Similarly, one could apply an approach like ours to sequences of edit operations. Anyway, this requires the definition of pattern catalogs and is therefore limited to application domains where such catalogs are available or requires the combination with pattern mining [323, 324] – an area that itself is rather active research than mature technology.

The work by Burgueno et al. [48] relies on knowledge extracted from textual documents to provide meaningful suggestions. Our approach is pure model completion and we do not include further information in the model completion setting. In this sense their approach can be understood as an extension of our approach that provides a completion in the form $C\colon \mathcal{T} \times \Sigma^* \to \mathcal{T}$. On the other hand, one can include other artifacts (e.g., requirements) and natural language information also as part of the model – which is done in the form of requirements diagrams for the SysML models of our Industry dataset. In this sense – when the additional information (e.g., requirements) that one wants to include in the model completion is fused with the to-be-completed models – our approach covers this "contextualized" model completion. We acknowledge the approach by Burgueno et al. [48] and also believe that a more explicit handling of different types of context has the potential to provide better targeted model completions. Using retrieval augmented generation, further context can easily be integrated in the generation. Unfortunately, we can not easily compare their approach to ours, because the dataset used in their evaluation is not available and in our datasets we do not have this additional context readily available, rendering a direct comparison difficult.

Both research streams (i.e., contextualized model completion and pattern-based model completion) seem to be promising concepts for further investigation and extension of the approach proposed in this work. Anyway, the more "combination" is used in the approach the more difficult will be to understand what effect is due to which design decision or technology (without conducting a sophisticated ablation study). The purpose of this work was to investigate the merits of LLM technology for model completion and therefore increase internal validity. Further combinations are intentionally left for future research.

Related but distinct tasks

While several related studies focus on related but different tasks, we will highlight a few examples here. Of course, mentioning all of them would exceed the scope of this work, and we hope the examples given here will clarify why some related work can not be directly compared to our approach, although similar on a high abstraction level. The work by Ohrndorf et al. [244], which specifically proposes a model repair approach rather than model completion. Their method generates repair proposals for inconsistencies introduced by incomplete editing processes. The approach focuses on examples in the revision history, in which constraints were, at some point, violated and subsequently fixed at a later point. While their method ensures constraints are preserved, it does not handle adding or changing functionality within the system, leaving the modeler to perform the actual modeling work. As we should not compare fixing syntactic errors in source to suggesting new source code, we can also not compare model repair to model completion. Gomes et al. [114] focus on creating and evolving a system domain model based on interactions in natural language from non-technical users. They utilize Natural Language Processing (NLP) to interpret the users' intents expressed in natural language and transfer these intents to commands the system can understand. This represents an entirely different task. They do not suggest functional changes to the model, but translate the user intentions to machine readable commands. Also the approaches by Kögel et al. [181] and Kushke et al. [193, 194] can be seen as different (altough very related) approaches. In principle, these approaches try to recommend complete patterns from partial patterns, which is then leveraged to perform a model completion. Finally, the approach by Burgueno et al. [48] recommends model completions given some additional context, while our approach requires only the model (change).

Domain-specific applications

There is a category of approaches focusing on specific domain languages [5, 129, 312]. Their approaches are limited to Simulink models, which is why we cannot apply them to our data.

Deng et al. [79] propose an approach focused on business process models, specifically BPMN. Their method is not transferable to other domains. They mine relationships among activity nodes from existing processes, store these relations as patterns

in a database, and then compare new processes with these patterns. This comparison recommends suitable activity nodes from the most matching patterns to assist in building a new process.

The need for a benchmarking infrastructure
Our analysis above underscores the current challenges faced by the research community. This lack of baselines and benchmarking infrastructure is a critical point. The field is currently in a state of developing the necessary infrastructure, and we are contributing to this effort, while appreciating previous contribution efforts.

Bibliography

[1] Felipe S. Abrahão, Santiago Hernández-Orozco, Narsis A. Kiani, Jesper Tegnér, and Hector Zenil. "Assembly Theory is an approximation to algorithmic complexity based on LZ compression that does not explain selection or evolution." In: *arXiv preprint* (2024). DOI: 10.48550/arXiv.2403.06629.

[2] Mathieu Acher and Jabier Martinez. "Generative AI for Reengineering Variants into Software Product Lines: An Experience Report." In: *Proceedings of the 27th ACM International Systems and Software Product Line Conference-Volume B*. 2023, pp. 57–66.

[3] Vlad Acreţoaie, Harald Störrle, and Daniel Strüber. "VMTL: A language for end-user model transformation." In: *Software & Systems Modeling* 17.4 (2018), pp. 1139–1167.

[4] Christoph Adami. "What is complexity?" In: *BioEssays* 24.10 (2002), pp. 1085–1094. DOI: 10.1002/bies.10192.

[5] Bhisma Adhikari, Eric J Rapos, and Matthew Stephan. "SimIMA: a virtual Simulink intelligent modeling assistant: Simulink intelligent modeling assistance through machine learning and model clones." In: *Software and Systems Modeling* (2023), pp. 1–28. DOI: 10.1007/s10270-023-01093-6.

[6] Charu C Aggarwal and Haixun Wang. "A survey of clustering algorithms for graph data." In: *Managing and mining graph data* (2010), pp. 275–301. DOI: 10.1007/978-1-4419-6045-0_9.

[7] Henning Agt-Rickauer, Ralf-Detlef Kutsche, and Harald Sack. "DoMoRe–a recommender system for domain modeling." In: *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Vol. 1. Setúbal: SciTePress. 2018, pp. 71–82. DOI: 10.5220/0006555700710082.

[8] Henning Agt-Rickauer, Ralf-Detlef Kutsche, and Harald Sack. "Automated recommendation of related model elements for domain models." In: *Model-driven Engineering and Software Development: 6th International Conference, MODELSWARD 2018, Funchal, Madeira, Portugal, January 22-24, 2018, Revised Selected Papers 6*. Springer. 2019, pp. 134–158.

[9] Aakash Ahmad, Muhammad Waseem, Peng Liang, Mahdi Fahmideh, Mst Shamima Aktar, and Tommi Mikkonen. "Towards human-bot collaborative software architecting with chatgpt." In: *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering*. 2023, pp. 279–285. DOI: 10.1145/3593434.3593468.

[10] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. "Unified pre-training for program understanding and generation." In: *arXiv* (2021). DOI: 10.48550/arXiv.2103.06333.

[11] Toufique Ahmed and Premkumar Devanbu. "Few-shot training LLMs for project-specific code-summarization." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. 2022, pp. 1–5. DOI: 10.1145/3551349.3559555.

[12] Leman Akoglu, Hanghang Tong, and Danai Koutra. "Graph based anomaly detection and description: a survey." In: *Data mining and knowledge discovery* 29 (2015), pp. 626–688. DOI: 10.1007/s10618-014-0365-y.

[13] Fernando Almeida. "Strategies to perform a mixed methods study." In: *European Journal of Education Studies* (2018).

[14] Lissette Almonte, Esther Guerra, Iván Cantador, and Juan de Lara. "Recommender systems in Model-driven Engineering." In: *Software and Systems Modeling* 21.1 (2022), pp. 249–280. DOI: 10.1007/s10270-021-00905-x.

[15] Abdullah M. Alshanqiti, Reiko Heckel, and Tamim Ahmed Khan. "Learning minimal and maximal rules from observations of graph transformations." In: *Electronic Communication of the European Association of Software Science and Technology* 47 (2012).

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

[17] Sven Apel, Florian Janda, Salvador Trujillo, and Christian Kästner. "Model superimposition in software product lines." In: *Proceedings of the International Conference on Theory and Practice of Model Transformations (ICMT)*. Springer. 2009, pp. 4–19. DOI: 10.1007/978-3-642-02408-5_2.

[18] Sven Apel, Christian Kastner, and Christian Lengauer. "Featurehouse: Language-independent, automated software composition." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE. 2009, pp. 221–231. DOI: 10.1109/ICSE.2009.5070523.

[19] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. "Henshin: Advanced concepts and tools for in-place EMF model transformations." In: *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Springer. 2010, pp. 121–135. DOI: 10.1007/978-3-642-16145-2_9.

[20] Thorsten Arendt and Gabriele Taentzer. "A tool environment for quality assurance based on the Eclipse Modeling Framework." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE/ACM, 2013, pp. 141–184. DOI: 10.1007/s10515-012-0114-7.

[21] Erik Arisholm, Lionel C Briand, Siw Elisabeth Hove, and Yvan Labiche. "The impact of UML documentation on software maintenance: An experimental evaluation." In: *Transactions on Software Engineering* 32.6 (2006), pp. 365–381. DOI: 10.1109/TSE.2006.59.

[22] Wesley KG Assunção, Roberto E Lopez-Herrejon, Lukas Linsbauer, Silvia R Vergilio, and Alexander Egyed. "Reengineering legacy applications into software product lines: a systematic mapping." In: *Empirical Software Engineering* 22 (2017), pp. 2972–3016.

[23] Iman Avazpour, John Grundy, and Lars Grunske. "Specifying model transformations by direct manipulation using concrete visual notations and interactive recommendations." In: *Journal of Visual Languages and Computing* 28 (2015), pp. 195–211. DOI: 10.1016/j.jvlc.2015.02.005.

[24] László Babai. "Graph Isomorphism in Quasipolynomial Time." In: *arXiv preprint* (2016).

[25] Viraj Bagal, Rishal Aggarwal, PK Vinod, and U Deva Priyakumar. "Molgpt: Molecular generation using a transformer-decoder model." In: *Journal of Chemical Information and Modeling* 62.9 (2021), pp. 2064–2076. DOI: 10.1021/acs.jcim.1c00600.

[26] Islem Baki and Houari Sahraoui. "Multi-step learning and adaptive search for learning complex model transformations from examples." In: *ACM Transactions on Software Engineering and Methodology* 25.3 (2016), pp. 1–36. DOI: 10.1145/2904904.

[27] Zsolt Balanyi and Rudolf Ferenc. "Mining design patterns from C++ source code." In: *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2003, pp. 305–314. DOI: 10.1109/ICSM.2003.1235436.

[28] Francesco Bariatti, Peggy Cellier, and Sébastien Ferré. "GraphMDL: Graph Pattern Selection Based on Minimum Description Length." In: *Advances in Intelligent Data Analysis XVIII*. Springer International Publishing, 2020, pp. 54–66. DOI: 10.1007/978-3-030-44584-3_5.

[29] Francesco Bariatti, Peggy Cellier, and Sébastien Ferré. "GraphMDL: Graph Pattern Selection Based on Minimum Description Length." In: *Advances in Intelligent Data Analysis XVIII*. Ed. by Michael R. Berthold, Ad Feelders, and Georg Krempl. Springer International Publishing, 2020, pp. 54–66. DOI: 10.1007/978-3-030-44584-3_5.

[30] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. "Scaling step-wise refinement." In: *Transactions on Software Engineering* 30.6 (2004), pp. 355–371. DOI: 10.1109/ICSE.2003.1201199.

[31] Gordon Baxter and Ian Sommerville. "Socio-technical systems: From design methods to systems engineering." In: *Interacting with Computers* 23.1 (2010). DOI: 10.1016/j.intcom.2010.07.003.

[32] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. "A Neural Probabilistic Language Model." In: *Advances in Neural Information Processing Systems*. MIT Press, 2000, pp. 932–938. DOI: 10.5555/944919.944966.

[33] Thorsten Berger, Marsha Chechik, Timo Kehrer, Manuel Wimmer, Thorsten Berger, and Manuel Wimmer. "Software Evolution in Time and Space: Unifying Version and Variability Management Edited by Executive Summary." In: *Dagstuhl Reports* 9.5 (2019), pp. 1–30.

[34] Thorsten Berger, Jan-Philipp Steghöfer, Tewfik Ziadi, Jacques Robin, and Jabier Martinez. "The state of adoption and the challenges of systematic variability management in industry." In: *Empirical Software Engineering* (2020). DOI: 10.1007/s10664-019-09787-6.

[35] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. "Viatra 3 : A Reactive Model Transformation Platform." In: *Proceedings of the International Conference on Model Transformations (ICMT)*. Springer. Springer, 2015. DOI: 10.1007/978-3-319-21155-8_8.

[36] Karima Berramla., El Abbassia Deba., Jiechen Wu., Houari Sahraoui., and Abou Benyamina. "Model Transformation by Example with Statistical Machine Translation." In: *Proceedings of the International Conference on Model-driven Engineering and Software Development (MODELSWARD)*. INSTICC. SciTePress, 2020, pp. 76–83. DOI: 10.5220/0009168200760083.

[37] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. "Formal foundation of consistent EMF model transformations by algebraic graph transformation." In: *Software and Systems Modeling* 11.2 (2012), pp. 227–250. DOI: 10.1007/s10270-011-0199-7.

[38] Paolo Bocciarelli and Andrea D'Ambrogio. "Chapter 14 - A model-driven method for the design-time performance analysis of service-oriented software systems." In: *Modeling and Simulation of Computer Networks and Systems*. Ed. by Mohammad S. Obaidat, Petros Nicopolitidis, and Faouzi Zarai. Boston: Morgan Kaufmann, 2015, pp. 425–450. ISBN: 978-0-12-800887-4. DOI: 10.1016/B978-0-12-800887-4.00014-6.

[39] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-driven software engineering in practice*. Morgan & Claypool Publishers, 2017.

[40] Frederick P. Brooks. *The mythical man-month – Essays on Software-Engineering*. Addison-Wesley, 1975.

[41] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. "An example is worth a thousend words: Composite operation modeling by-example." In: *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*. ACM, 2009, pp. 271–285.

[42] Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry, and Yves Le Traon. "Metamodel-based test generation for model transformations: an algorithm and a tool." In: *Proceedings of International Sympoisum on Software Reliability Engineering (ISSRE)*. 2006, pp. 85–94. DOI: 10.1109/ISSRE.2006.27.

[43] Cédric Brun and Alfonso Pierantonio. "Model differences in the eclipse modeling framework." In: *UPGRADE, The European Journal for the Informatics Professional* 9.2 (2008), pp. 29–34.

[44] Greg Brunet, Marsha Chechik, Steve Easterbrook, Shiva Nejati, Nan Niu, and Mehrdad Sabetzadeh. "A manifesto for model merging." In: *Proceedings of the 2006 international workshop on Global integrated model management*. 2006, pp. 5–12. DOI: 10.1145/1138304.1138307.

[45] Antonio Bucchiarone, Jordi Cabot, Richard F Paige, and Alfonso Pierantonio. "Grand challenges in Model-driven Engineering: an analysis of the state of the research." In: *Software and Systems Modeling* 19 (2020), pp. 5–13.

[46] Alexandru Burdusel, Steffen Zschaler, and Daniel Strüber. "MDEOptimiser: A search based model engineering tool." In: *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS): Companion Proceedings*. ACM, 2018, pp. 12–16. DOI: 10.1145/3270112.3270130.

[47] Loli Burgueño, Jordi Cabot, Shuai Li, and Sébastien Gérard. "A generic LSTM neural network architecture to infer heterogeneous model transformations." In: *Software and Systems Modeling* 21.1 (2022), pp. 139–156. DOI: 10.1007/s10270-021-00893-y.

[48] Loli Burgueño, Robert Clarisó, Sébastien Gérard, Shuai Li, and Jordi Cabot. "An NLP-based architecture for the autocompletion of partial domain models." In: *Proceedings of the International Conference on Advanced Information Systems Engineering*. Springer. 2021, pp. 91–106. DOI: 10.1007/978-3-030-79382-1_6.

[49] Loli Burgueño, Jordi Cabot, and Sébastien Gérard. "An LSTM-Based Neural Network Architecture for Model Transformations." In: *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 2019, pp. 294–299. DOI: 10.1109/MODELS.2019.00013.

[50] Jordi Cabot, Robert Clarisó, Marco Brambilla, and Sébastien Gérard. "Cognifying model-driven software engineering." In: *Software Technologies: Applications and Foundations*. Springer. 2018, pp. 154–160. DOI: 10.1007/978-3-319-74730-9_13.

[51] Javier Cámara, Javier Troya, Lola Burgueño, and Antonio Vallecillo. "On the assessment of generative AI in modeling tasks: an experience report with ChatGPT and UML." In: *Software and Systems Modeling* (2023), pp. 1–13. DOI: 10.1007/s10270-023-01105-5.

[52] Jaime Carbonell and Jade Goldstein. "The Use of MMR, Diversity-Based Reranking for Reordering Documents and Producing Summaries." In: *Proceedings of the International Conference on Research and Development in Information Retrieval*. New York, NY, USA: ACM, 1998, 335—336. DOI: 10.1145/290941.291025.

[53] Jessie Carbonnel, Marianne Huchard, and Clémentine Nebut. "Modeling equivalence classes of feature models with concept lattices to assist their extraction from product descriptions." In: *Journal of Systems and Software* 152 (2019), pp. 1–23.

[54] Sean B. Carroll. "Chance and necessity: the evolution of morphological complexity and diversity." In: *Nature* 409.6821 (2001), pp. 1102–1109. DOI: 10.1038/35059227.

[55] CENELEC/EN. *50126: Railway applications-The specification and demonstration of Reliability*. Tech. rep. CENELEC/EN, 2001.

[56] CENELEC/EN. *50129: Railway application–Communications, signaling and processing systems–Safety related electronic systems for signaling*. Tech. rep. 2003.

[57] CENELEC/EN. *50128: Railway applications-Communication, signalling and processing systems-Software for railway control and protection systems*. Tech. rep. CENELEC/EN, 2012.

[58] Meriem Ben Chaaben, Lola Burgueño, and Houari Sahraoui. "Towards using few-shot prompt learning for automating model completion." In: *Proceedings of the International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE. 2023, pp. 7–12. DOI: 10.1109/ICSE-NIER58687.2023.00008.

[59] Kaylea Champion, Sejal Khatri, and Benjamin Mako Hill. "Qualities of Quality: A Tertiary Review of Software Quality Measurement Research." In: *CoRR* abs/2107.13687 (2021). DOI: 10.48550/arXiv.2107.13687.

[60] Ned Chapin, Joanne E Hale, Khaled Md Khan, Juan F Ramil, and Wui-Gee Tan. "Types of software evolution and software maintenance." In: *Journal of software maintenance and evolution: Research and Practice* 13.1 (2001), pp. 3–30. DOI: 10.1002/smr.220.

[61] Lingjiao Chen, Matei Zaharia, and James Zou. "How is ChatGPT's behavior changing over time?" In: *arXiv* (2023). DOI: 10.48550/arXiv.2307.09009.

[62] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. "Evaluating large language models trained on code." In: *arXiv* (2021). DOI: 10.48550/arXiv.2107.03374.

[63] Zhengdao Chen, Lei Chen, Soledad Villar, and Joan Bruna. "Can graph neural networks count substructures?" In: *Advances in neural information processing systems* 33 (2020), pp. 10383–10395. DOI: 10.5555/3495724.3496595.

[64] Zhikai Chen, Haitao Mao, Hang Li, Wei Jin, Hongzhi Wen, Xiaochi Wei, Shuaiqiang Wang, Dawei Yin, Wenqi Fan, Hui Liu, et al. "Exploring the potential of large language models (llms) in learning on graphs." In: *ACM SIGKDD Explorations Newsletter* 25.2 (2024). DOI: 10.1145/3655103.3655110.

[65] Noam Chomsky. "Three models for the description of language." In: *IRE Transactions on information theory* 2.3 (1956), pp. 113–124.

[66] Soudip Roy Chowdhury, Florian Daniel, and Fabio Casati. "Recommendation and Weaving of Reusable Mashup Model Patterns for Assisted Development." In: *ACM Transactions on Internet Technology* 14.2-3 (2014). DOI: 10.1145/2663500.

[67] Tsigkanos Christos, Rani Pooja, Müller Sebastian, and Kehrer Timo. "Large language models: the next frontier for variable discovery within metamorphic testing?" In: *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2023.

[68] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. "An Empirical Study on the Usage of Transformer Models for Code Completion." In: *Transactions on Software Engineering* 48.12 (2022), pp. 4818–4837. DOI: 10.1109/TSE.2021.3128234.

[69] *Clarifying concepts: MBE vs MDE vs MDD vs MDA*. https://modeling-languages.com/clarifying-concepts-mbe-vs-mde-vs-mdd-vs-mda/. Accessed: 2023-04-18.

[70] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Boston, 2002.

[71] Benoit Combemale, Jorg Kienzle, Gunter Mussbacher, Hyacinth Ali, Daniel Amyot, Mojtaba Bagherzadeh, Edouard Batot, Nelly Bencomo, Benjamin Benni, Jean-Michel Bruel, et al. "A Hitchhiker's Guide to Model-driven Engineering for Data-Centric Systems." In: *IEEE Software* 38.4 (2020), pp. 71–84. DOI: 10.1109/MS.2020.2995125.

[72] Diane J Cook and Lawrence B Holder. *Mining graph data*. John Wiley & Sons, 2006. DOI: 10.1002/0470073047.

[73] James B Dabney and Thomas L Harman. *Mastering simulink*. Vol. 230. Pearson/Prentice Hall Upper Saddle River, 2004. DOI: 10.5555/861366.

[74] Jason Dagit and Matthew J. Sottile. "Identifying change patterns in software history." In: *Proceedings of the International workshop on Document Changes: Modeling, Detection, Storage and Visualization*. CEUR-WS.org, 2013.

[75] Carlos Diego Nascimento Damasceno and Daniel Strüber. "Quality guidelines for research artifacts in Model-driven Engineering." In: *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE. 2021, pp. 285–296. DOI: 10.1109/MODELS50736.2021.00036.

[76] Charles Darwin. "On the origin of species by means of natural selection, or preservation of favoured races in the struggle for life." In: *London: John Murray* (1859).

[77] Nicola De Cao and Thomas Kipf. "MolGAN: An implicit generative model for small molecular graphs." In: *arXiv preprint arXiv:1805.11973* (2018).

[78] Deva Kumar Deeptimahanti and Ratna Sanyal. "Semi-automatic generation of UML models from natural language requirements." In: ACM, 2011, 165—174. DOI: 10.1145/1953355.1953378.

[79] Shuiguang Deng, Dongjing Wang, Ying Li, Bin Cao, Jianwei Yin, Zhaohui Wu, and Mengchu Zhou. "A recommendation system to facilitate business process modeling." In: *IEEE transactions on cybernetics* 47.6 (2016), pp. 1380–1394. DOI: 10.1109/TCYB.2016.2545688.

[80] Juri Di Rocco, Davide Di Ruscio, Claudio Di Sipio, Phuong T Nguyen, and Alfonso Pierantonio. "MemoRec: a recommender system for assisting modelers in specifying metamodels." In: *Software and Systems Modeling* 22.1 (2023), pp. 203–223. DOI: 10.1007/s10270-022-00994-2.

[81] Juri Di Rocco, Claudio Di Sipio, Phuong T Nguyen, Davide Di Ruscio, and Alfonso Pierantonio. "Finding with nemo: a recommender system to forecast the next modeling operations." In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*. 2022, pp. 154–164.

[82] Claudio Di Sipio, Juri Di Rocco, Davide Di Ruscio, and Phuong T Nguyen. "MORGAN: a modeling recommender system based on graph kernel." In: *Software and Systems Modeling* (2023), pp. 1–23. DOI: 10.1007/s10270-023-01102-8.

[83] Surnjani Djoko. "Substructure discovery using minimum description length principle and background knowledge." In: *Proceedings of the National Conference on Artificial Intelligence* 2 (1994), p. 1442.

[84] Jing Dong, Yajing Zhao, and Tu Peng. "A review of design pattern mining techniques." In: *International Journal of Software Engineering and Knowledge Engineering* 19.6 (2009), pp. 823–855. DOI: 10.1142/S021819400900443X.

[85] Louise Doyle, Anne-Marie Brady, and Gobnait Byrne. "An overview of mixed methods research." In: *Journal of research in nursing* 14.2 (2009), pp. 175–185.

[86] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. "An exploratory study of cloning in industrial software product lines." In: *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2013, pp. 25–34. DOI: 10.1109/CSMR.2013.13.

[87] Vijay Prakash Dwivedi and Xavier Bresson. "A Generalization of Transformer Networks to Graphs." In: *arXiv preprint* (2021). DOI: 10.48550/arXiv.2012.09699.

[88] Jorge Echeverría, Francisca Pérez, José Ignacio Panach, and Carlos Cetina. "An empirical study of performance using Clone & Own and Software Product Lines in an industrial context." In: *Information and Software Technology* 130 (2021). DOI: 10.1016/j.infsof.2020.106444.

[89] H. Ehrig, M. Pfender, and H. J. Schneider. "Graph-grammars: An algebraic approach." In: *Annual Symposium on Switching and Automata Theory*. 1973, pp. 167–180. DOI: 10.1109/SWAT.1973.11.

[90] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. "Fundamental theory for typed attributed graph transformation." In: *Lecture Notes in Computer Science*. Vol. 3256. 2004, pp. 161–177. DOI: 10.1007/978-3-540-30203-2_13.

[91] Karsten Ehrig, Claudia Ermel, Stefan Hänsgen, and Gabriele Taentzer. "Generation of visual editors as Eclipse plug-ins." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2005, pp. 134–143. DOI: 10.1145/1101908.1101930.

[92] Akil Elkamel, Mariem Gzara, and Hanêne Ben-Abdallah. "An UML class recommender system for software design." In: *Proceedings of the International Conference of Computer Systems and Applications (AICCSA)*. IEEE. 2016, pp. 1–8. DOI: 10.1109/AICCSA.2016.7945659.

[93] Geoffrey Elliott. *Global business information technology: an integrated systems approach*. Pearson Education, 2004.

[94] Tzilla Elrad, Omar Aldawud, and Atef Bader. "Aspect-oriented modeling: Bridging the gap between implementation and design." In: *International Conference on Generative Programming and Component Engineering*. Springer. 2002, pp. 189–201. DOI: 10.1007/3-540-45821-2_12.

[95] Ameni ben Fadhel, Marouane Kessentini, Philip Langer, and Manuel Wimmer. "Search-based detection of high-level model changes." In: *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 212–221. DOI: 10.1109/ICSM.2012.6405274.

[96] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. "Codebert: A pre-trained model for programming and natural languages." In: *arXiv* (2020). DOI: 10.48550/arXiv.2002.08155.

[97] Rudolf Ferenc, Arpad Beszedes, Lajos Fulop, and Janos Lele. "Design pattern mining enhanced by machine learning." In: *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2005, pp. 295–304. DOI: 10.1109/ICSM.2005.40.

[98] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. "A vision for enhancing clone-and-own with systematic reuse for developing software variants." In: *Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft fur Informatik (GI)* P252 (2016), pp. 95–96.

[99] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. "Enhancing clone-and-own with systematic reuse for developing software variants." In: *2014 IEEE International conference on software maintenance and evolution*. IEEE. 2014, pp. 391–400.

[100] W Tecumseh Fitch. *The evolution of language*. Cambridge University Press, 2010.

[101] John Fitzgerald and Peter Larsen. "Modeling systems. Practical tools and techniques in software development. With CD-ROM." In: *Modeling Systems: Practical Tools and Techniques in Software Development* (Jan. 2009). DOI: 10.1017/CBO9780511626975.

[102] Walter Fontana. *Algorithmic chemistry*. Tech. rep. Los Alamos National Lab., NM (USA), 1990.

[103] David Foster. *Generative deep learning*. " O'Reilly Media, Inc.", 2022.

[104] Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[105] Robert France and Bernhard Rumpe. "Model-driven development of complex software: A research roadmap." In: *Proceedings of the Conference on Future of Software Engineering (FOSE)*. IEEE. 2007, pp. 37–54. DOI: 10.1109/FOSE.2007.14.

[106] Erich Gamma, Ralph Johnson, Richard Helm, Ralph E Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Prentice Hall, 1995.

[107] Michael R Garey and David S Johnson. *Computers and intractability*. Vol. 174. freeman San Francisco, 1979.

[108] Tarun Garg, Kaushik Roy, and A. Sheth. "Can Language Models Capture Graph Semantics? From Graphs to Language Model and Vice-Versa." In: *arXiv* (2022). DOI: 10.48550/arXiv.2206.09259.

[109] David Garlan, Robert Allen, and John Ockerbloom. "Architectural mismatch or why it's hard to build systems out of existing parts." In: *Proceedings of the 17th international conference on Software engineering*. 1995, pp. 179–185. DOI: 10.1145/225014.225031.

[110] Thomas Gärtner. "A survey of kernels for structured data." In: *ACM SIGKDD explorations newsletter* 5.1 (2003), pp. 49–58.

[111] Sinem Getir, Lars Grunske, André van Hoorn, Timo Kehrer, Yannic Noller, and Matthias Tichy. "Supporting semi-automatic co-evolution of architecture and fault tree models." In: *Journal of Systems and Software* 142 (2018), pp. 115–135. DOI: 10.1016/j.jss.2018.04.001.

[112] Adnane Ghannem, Marouane Kessentini, Mohammad Salah Hamdi, and Ghizlane El Boussaidi. "Model refactoring by example: A multi-objective search based software engineering approach." In: *Journal of Software: Evolution and Process* 30.4 (2018), pp. 1–20.

[113] Michael W. Godfrey and Daniel M. German. "The past, present, and future of software evolution." In: *Frontiers of Software Maintenance*. 2008, pp. 129–138. DOI: 10.1109/FOSM.2008.4659256.

[114] Anderson Gomes and Paulo Henrique M Maia. "DoME: An Architecture for Domain Model Evolution at Runtime Using NLP." In: *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*. 2023, pp. 186–195.

[115] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative adversarial networks." In: *Communications of the ACM* 63.11 (2020). DOI: 10.1145/3422622.

[116] J. Gorodkin. "Comparing two K-category assignments by a K-category correlation coefficient." In: *Computational Biology and Chemistry* 28.5 (2004). DOI: 10.1016/j.compbiolchem.2004.09.006.

[117] Peter Grunwald. "A tutorial introduction to the minimum description length principle." In: *arXiv preprint* (2004). DOI: 10.48550/arXiv.math/0406077.

[118] Peter D Grünwald and Abhijit Grunwald. *The minimum description length principle*. MIT press, 2007. DOI: 10.7551/mitpress/4643.001.0001.

[119] Peter D Grünwald and Paul MB et al. Vitányi. "Algorithmic information theory." In: *Handbook of the Philosophy of Information* (2008), pp. 281–320.

[120] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. "Program synthesis." In: *Foundations and Trends® in Programming Languages* 4.1-2 (2017), pp. 1–119. DOI: 10.1561/2500000010.

[121] William L Hamilton, Rex Ying, and Jure Leskovec. "Representation learning on graphs: Methods and applications." In: *arXiv preprint* (2017). DOI: 10.48550/arXiv.1709.05584.

[122] Zellig S. Harris. "Distributional Structure." In: *WORD* 10.2-3 (1954), pp. 146–162. DOI: 10.1080/00437956.1954.11659520.

[123] GM Harshvardhan, Mahendra Kumar Gourisaria, Manjusha Pandey, and Siddharth Swarup Rautaray. "A comprehensive survey and analysis of generative models in machine learning." In: *Computer Science Review* 38 (2020). DOI: 10.1016/j.cosrev.2020.100285.

[124] David Haussler et al. *Convolution kernels on discrete structures*. Tech. rep. Citeseer, 1999.

[125] Robert M Hazen, Peter C Burns, H James Cleaves, Robert T Downs, Sergey V Krivovichev, and Michael L Wong. "Molecular assembly indices of mineral heteropolyanions: some abiotic molecules are as complex as large biomolecules." In: *Journal of the Royal Society Interface* 21.211 (2024), p. 20230632. DOI: 10.1098/rsif.2023.0632.

[126] Regina Hebig, Djamel Eddine Khelladi, and Reda Bendraou. "Approaches to co-evolution of metamodels and models: A survey." In: *Transactions on Software Engineering* 43.5 (2017), pp. 396–414. DOI: 10.1109/TSE.2016.2610424.

[127] Ábel Hegedüs, Ákos Horváth, István Ráth, Moisés Castelo Branco, and Dániel Varró. "Quick fix generation for DSMLs." In: *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2011, pp. 17–24. DOI: 10.1109/VLHCC.2011.6070373.

[128] Wolfgang Heider, Roman Froschauer, Paul Grünbacher, Rick Rabiser, and Deepak Dhungana. "Simulating evolution in model-based product line engineering." In: *Information and Software Technology* 52.7 (2010), pp. 758–769. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2010.03.007.

[129] Lars Heinemann. "Facilitating reuse in model-based development with context-dependent model element recommendations." In: *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*. IEEE. 2012, pp. 16–20. DOI: 10.1109/RSSE.2012.6233402.

[130] Martin Henkel and Janis Stirna. "Pondering on the Key Functionality of Model Development Tools: The Case of Mendix." In: *Perspectives in Business Informatics Research*. Springer. 2010, pp. 146–160.

[131] Markus Herrmannsdoerfer and Maximilian Koegel. "Towards a generic operation recorder for model evolution." In: 2010, pp. 76–81. DOI: 10.1145/1826147.1826161.

[132] Markus Herrmannsdoerfer, Sander Vermolen, and Guido Wachsmuth. "An extensive catalog of operators for the coupled evolution of metamodels and models." In: *Software Language Engineering*. ACM, 2010, pp. 163–182. DOI: 10.1007/978-3-642-19440-5_10.

[133] Geoffrey E. Hinton. "Learning distributed representations of concepts." In: *Proceedings of the Annual Conference of the Cognitive Science Society*. Vol. 1. Amherst, MA, 1986.

[134] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. "Systematically deriving domain-specific transformation languages." In: *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*. ACM/IEEE, 2015, pp. 136–145. DOI: 10.1109/MODELS.2015.7338244.

[135] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators." In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: 10.1016/0893-6080(89)90020-8.

[136] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. "Strategies for pre-training graph neural networks." In: *arXiv preprint* (2019).

[137] Ningyuan Teresa Huang and Soledad Villar. "A short tutorial on the weisfeiler-lehman test and its variants." In: *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2021, pp. 8533–8537. DOI: 10.1109/ICASSP39728.2021.9413523.

[138] Rasha Ahmad Husein, Hala Aburajouh, and Cagatay Catal. "Large language models for code completion: A systematic literature review." In: *Computer Standards and Interfaces* 92 (2025), p. 103917. DOI: 10.1016/j.csi.2024.103917.

[139] John Hutchinson, Jon Whittle, and Mark Rouncefield. "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure." In: *Science of Computer Programming* 89 (2014), pp. 144–161.

[140] IEC. *61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*. Tech. rep. IEC, 2010.

[141] No Magic Inc. *MagicDraw website*. Sept. 2021. URL: https://www.magicdraw.com/.

[142] Ludovico Iovino, Angela Barriga Rodriguez, Adrian Rutle, and Rogardt Heldal. "Model repair with quality-based reinforcement learning." In: (2020). DOI: doi:10.5381/jot.2020.19.2.a17.

[143] May ISO. *Systems and software engineering–architecture description*. Tech. rep. ISO/IEC/IEEE 42010, 2011.

[144] Johannes Jaeger. "Assembly Theory: What It Does and What It Does Not Do." In: *Journal of Molecular Evolution* 92.2 (2024), pp. 87–92. DOI: 10.1007/s00239-024-10163-2.

[145] Mario Janke and Patrick Mäder. "Graph Based Mining of Code Change Patterns from Version Control Commits." In: *Transactions on Software Engineering* (2020). DOI: 10.1109/TSE.2020.3004892.

[146] Anton Jansen and Jan Bosch. "Software architecture as a set of architectural design decisions." In: *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*. IEEE. 2005, pp. 109–120.

[147] Praveen Jayaraman, Jon Whittle, Ahmed M Elkhodary, and Hassan Gomaa. "Model composition in product lines and feature interaction detection using critical pair analysis." In: *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Springer. 2007, pp. 151–165. DOI: 10.1007/978-3-540-75209-7_11.

[148] F. Jelinek and R.L. Mercer. "Interpolated estimation of Markov source parameters from sparse data." In: *Proceedings of the Workshop on Pattern Recognition in Practice*. 1980.

[149] Hans Peter Jepsen, Jan Gaardsted Dall, and Danilo Beuche. "Minimally Invasive Migration to Software Product Lines." In: *Proceedings of the Software Product Line Conference (SPLC)*. 2007, pp. 203–211. DOI: 10.1109/SPLINE.2007.30.

[150] Kevin Jesse, Toufique Ahmed, Premkumar T Devanbu, and Emily Morgan. "Large Language Models and Simple, Stupid Bugs." In: *arXiv* (2023). DOI: 10.1109/MSR59073.2023.00082.

[151] Chuntao Jiang, Frans Coenen, and Michele Zito. "A survey of frequent subgraph mining algorithms." In: *Knowledge Engineering Review* 28.1 (2013), pp. 75–105. DOI: 10.1017/S0269888912000331.

[152] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. "A Survey on Large Language Models for Code Generation." In: *arXiv* (2024). DOI: 10.48550/arXiv.2406.00515.

[153] Frédéric Jouault and Ivan Kurtev. "Transforming models with ATL." In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2005, pp. 128–138. DOI: 10.1007/11663430_14.

[154] Nafiseh Kahani, Mojtaba Bagherzadeh, James R Cordy, Juergen Dingel, and Daniel Varró. "Survey and classification of model transformation tools." In: *Software & Systems Modeling* 18.4 (2019), pp. 2361–2397. DOI: 10.1007/s10270-018-0665-6.

[155] Gerti Kappel, Philip Langer, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. "Model transformation by-example: A survey of the first wave." In: *Conceptual Modeling and Its Theoretical Foundations - Essays Dedicated to Bernhard Thalheim on the Occasion of His 60th Birthday*. Springer, 2012, pp. 197–215. DOI: 10.1007/978-3-642-28279-9_15.

[156] Cory J Kapser and Michael W Godfrey. ""Cloning considered harmful" considered harmful: patterns of cloning in software." In: *Empirical Software Engineering* 13.6 (2008), pp. 645–692.

[157] Christian Kästner, Salvador Trujillo, and Sven Apel. "Visualizing Software Product Line Variabilities in Source Code." In: *Proceedings of the Software Product Line Conference (SPLC)*. 2008, pp. 303–312.

[158] Stuart Kauffman and Philip Clayton. "On emergence, agency, and organization." In: *Biology and Philosophy* 21.4 (2006), pp. 501–521.

[159] Timo Kehrer. "Calculation and Propagation of Model Changes based on User-Level Edit Operations: A Foundation for Version and Variant Management in Model-driven Engineering." PhD thesis. University of Siegen, 2015.

[160] Timo Kehrer, Abdullah M Alshanqiti, and Reiko Heckel. "Automatic inference of rule-based specifications of complex in-place model transformations." In: *Proceedings of the International Conference on Theory and Practice of Model Transformations (ICMT)*. Springer, 2017, pp. 92–107. DOI: 10.1007/978-3-319-61473-1_7.

[161] Timo Kehrer, Udo Kelter, Manuel Ohrndorf, and Tim Sollbach. "Understanding model evolution through semantically lifting model differences with SiLift." In: *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 638–641. DOI: 10.1109/ICSM.2012.6405342.

[162] Timo Kehrer, Udo Kelter, Pit Pietsch, and Maik Schmidt. "Adaptability of model comparison tools." In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE. 2012, pp. 306–309. DOI: 10.1145/2351676.2351731.

[163] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. "A rule-based approach to the semantic lifting of model differences in the context of model versioning." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM/IEEE, 2011, pp. 163–172. DOI: 10.1109/ASE.2011.6100050.

[164] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. "Consistency-preserving edit scripts in model versioning." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. 2013, pp. 191–201. DOI: 10.1109/ASE.2013.6693079.

[165] Timo Kehrer, Michaela Rindt, Pit Pietsch, and Udo Kelter. "Generating Edit Operations for Profiled UML Models." In: *MoDELS Workshop on Models and Evolution (ME@MoDELS)*. Citeseer. 2013, pp. 30–39. DOI: 10.1.1.402.8572.

[166] Timo Kehrer, Gabriele Taentzer, Michaela Rindt, and Udo Kelter. "Automatically deriving the specification of model editing operations from metamodels." In: *Proceedings of the International Conference on Theory and Practice of Model Transformations (ICMT)*. Vol. 9765. 2016, pp. 173–188. DOI: 10.1007/978-3-319-42064-6_12.

[167] Timo Kehrer, Thomas Thüm, Alexander Schultheiß, and Paul Maximilian Bittner. "Bridging the Gap Between Clone-and-Own and Software Product Lines." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2021, pp. 21–25. DOI: 10.1109/ICSE-NIER52604.2021.00013.

[168] R Keller, T Eger, CM Eckert, and PJ Clarkson. "Visualising change propagation." In: *Proceedings of the International Conference on Engineering Design*. 2005. DOI: abs/10.3316/informit.375337868449067.

[169] Stuart Kent. "Model driven engineering." In: *International conference on integrated formal methods*. Springer. 2002, pp. 286–298. DOI: 10.1007/3-540-47884-1_16.

[170] Nikhil S. Ketkar, Lawrence B. Holder, and Diane J. Cook. "Subdue: Compression-Based Frequent Pattern Discovery in Graph Data." In: *Proceedings of the 1st International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations*. ACM, 2005, 71–76. DOI: 10.1145/1133905.1133915.

[171] Amal Khalil and Juergen Dingel. "Supporting the evolution of UML models in model driven software development: a survey." In: *School of Computing, Queen's University, Ontario* (2013).

[172] Djamel Eddine Khelladi, Regina Hebig, Reda Bendraou, Jacques Robin, and Marie-Pierre Gervais. "Detecting complex changes and refactorings during (Meta)model evolution." In: *Information Systems* 62 (2016), pp. 220–241. DOI: 10.1016/j.is.2016.05.002.

[173] Diederik P Kingma. "Adam: A method for stochastic optimization." In: *arXiv preprint* (2014). DOI: 10.1145/1830483.1830503.

[174] Diederik P Kingma and Max Welling. "Auto-encoding variational bayes." In: *arXiv preprint* (2013). DOI: 10.48550/arXiv.1312.6114.

[175] Thomas N Kipf and Max Welling. "Variational graph auto-encoders." In: *arXiv preprint* (2016). DOI: 10.48550/arXiv.1611.07308.

[176] T.N. Kipf and M. Welling. "Semi-Supervised Classification with Graph Convolutional Networks." In: *Proceedings of the International Conference on Learning Representations (ICLR)*. 2017. DOI: 10.1145/3295222.3295349.

[177] T. Klein, U. A. Nickel, J. Niere, and A. Zündorf. *From UML to Java And Back Again*. Tech. rep. University of Paderborn, 1999.

[178] Maximilian Koegel, Jonas Helming, and Stephan Seyboth. "Operation-based conflict detection and resolution." In: *Proceedings of the ICSE Workshop on Comparison and Versioning of Software Models*. IEEE. 2009, pp. 43–48. DOI: 10.1109/CVSM.2009.5071721.

[179] Stefan Kögel. "Recommender system for model driven software development." In: *Proceedings of the Joint Meeting on Foundations of Software Engineering*. 2017, pp. 1026–1029. DOI: 10.1145/3106237.3119874.

[180] Stefan Kögel, Raffaela Groner, and Matthias Tichy. "Automatic change recommendation of models and meta models based on change histories." In: *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*. ACM/IEEE, 2016, pp. 14–19.

[181] Stefan Kögel, Raffaela Groner, and Matthias Tichy. "Automatic Change Recommendation of Models and Meta Models Based on Change Histories." In: *ME@ MoDELS*. 2016, pp. 14–19.

[182] Dimitrios S Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard Paige. "Different Models for Model Matching: An analysis of approaches to support model differencing." In: *Proceedings of the ICSE Workshop on Comparison and Versioning of Software Models*. IEEE. 2009, pp. 1–6. DOI: 10.1109/CVSM.2009.5071714.

[183] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. "Merging models with the epsilon merging language (eml)." In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2006, pp. 215–229. DOI: 10.1007/11880240_16.

[184] Dimitrios S Kolovos, Louis M Rose, Saad Bin Abid, Richard F Paige, Fiona AC Polack, and Goetz Botterweck. "Taming EMF and GMF using model transformation." In: *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Springer. 2010, pp. 211–225. DOI: 10.1007/978-3-642-16145-2_15.

[185] Dimitrios S Kolovos, Louis M Rose, Nicholas Matragkas, Richard F Paige, Esther Guerra, Jesús Sánchez Cuadrado, Juan De Lara, István Ráth, Dániel Varró, Massimo Tisi, et al. "A research roadmap towards achieving scalability in model driven engineering." In: *Proceedings of the Workshop on Scalability in Model Driven Engineering*. 2013, pp. 1–10. DOI: 10.1145/2487766.2487768.

[186] Dimitris Kolovos. "What is Model-driven Engineering?" In: *Codebots* (2021). Accessed: 2024-04-28.

[187] Philippe B Kruchten. "The 4+ 1 view model of architecture." In: *IEEE software* 12.6 (1995), pp. 42–50. DOI: 10.1109/52.469759.

[188] Charles Krueger and Paul Clements. "Systems and software product line engineering." In: *Encyclopedia of Software Engineering* 2 (2013), pp. 1–14.

[189] Charles W. Krueger. "Software Reuse." In: *ObjectWorld Conference* (1993). DOI: 10.1145/130844.130856.

[190] Jacob Krüger and Thorsten Berger. "An empirical analysis of the costs of clone-and platform-oriented software reuse." In: *Proceedings of the Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2020, pp. 432–444. DOI: 10.1145/3368089.3409684.

[191] Thomas S Kuhn. *The structure of scientific revolutions*. Vol. 962. University of Chicago press Chicago, 1997.

[192] Marco Kuhrmann, Philipp Diebold, Jürgen Münch, et al. "Hybrid Software and System Development in Practice: Waterfall, Scrum, and Beyond." In: *Proceedings of the International Conference on Software and System Process*. 2017, pp. 30–39. DOI: 10.1145/3084100.3084104.

[193] Tobias Kuschke and Patrick Mäder. "RapMOD—In Situ Auto-Completion for Graphical Models." In: *Proceedings of the International Conference on Software Engineering (ICSE): Companion Proceedings*. IEEE. 2017, pp. 303–304. DOI: 10.1109/ICSE-C.2017.119.

[194] Tobias Kuschke, Patrick Mäder, and Patrick Rempel. "Recommending auto-completions for software modeling activities." In: *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Springer. 2013, pp. 170–186. DOI: 10.1007/978-3-642-41533-3_11.

[195] Matt J Kusner, Brooks Paige, and José Miguel Hernández-Lobato. "Grammar variational autoencoder." In: *International conference on machine learning*. PMLR. 2017, pp. 1945–1954. DOI: 10.48550/arXiv.1703.01925.

[196] Youngchun Kwon, Jiho Yoo, Youn-Suk Choi, Won-Joon Son, Dongseon Lee, and Seokho Kang. "Efficient learning of non-autoregressive graph variational autoencoders for molecular graph generation." In: *Journal of Cheminformatics* 11.1 (2019), pp. 1–10.

[197] Philip Langer, Manuel Wimmer, Petra Brosch, Markus Herrmannsdörfer, Martina Seidl, Konrad Wieland, and Gerti Kappel. "A posteriori operation detection in evolving software models." In: *Journal of Systems and Software* 86.2 (2013), pp. 551–566. DOI: 10.1016/j.jss.2012.09.037.

[198] Kristen LeFevre and Evimaria Terzi. "GraSS: Graph Structure Summarization." In: *Proceedings of the SIAM International Conference on Data Mining (SDM)*. 2010, pp. 454–465. DOI: 10.1137/1.9781611972801.40.

[199] Dean Leffingwell. *Scaling Software Agility: Best Practices for Large Enterprises*. Pearson Education, 2007.

[200] Meir M Lehman, Juan F Ramil, Paul D Wernick, Dewayne E Perry, and Wladyslaw M Turski. "Metrics and laws of software evolution-the nineties view." In: *Proceedings Fourth International Software Metrics Symposium*. IEEE. 1997, pp. 20–32. DOI: 10.1109/METRIC.1997.637156.

[201] M.M. Lehman. "Programs, life cycles, and laws of software evolution." In: *Proceedings of the IEEE* 68.9 (1980), pp. 1060–1076. DOI: 10.1109/PROC.1980.11805.

[202] Timothy C. Lethbridge, Janice Singer, and Andrew Forward. "How software engineers use documentation: the state of the practice." In: *IEEE Software* 20.6 (2003), pp. 35–39. DOI: 10.1109/MS.2003.1241364.

[203] David A Levin and Yuval Peres. *Markov chains and mixing times*. Vol. 107. American Mathematical Soc., 2017.

[204] Ming Li and Paul M. B. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications, 4th Edition*. Texts in Computer Science. Springer, 2019. DOI: 10.1007/978-3-030-11298-1.

[205] Ying Li, Bin Cao, Lida Xu, Jianwei Yin, Shuiguang Deng, Yuyu Yin, and Zhaohui Wu. "An efficient recommendation method for improving business process modeling." In: *IEEE Transactions on Industrial Informatics* 10.1 (2013), pp. 502–513. DOI: 10.1109/TII.2013.2258677.

[206] Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. "Learning deep generative models of graphs." In: *arXiv preprint* (2018). DOI: 10.48550/arXiv.1803.03324.

[207] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. "A classification of variation control systems." In: *ACM SIGPLAN Notices* 52.12 (2017), pp. 49–62. DOI: 10.1145/3136040.3136054.

[208] Xin Liu, Haojie Pan, Mutian He, Yangqiu Song, Xin Jiang, and Lifeng Shang. "Neural subgraph isomorphism counting." In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2020, pp. 1959–1969.

[209] José Antonio Hernández López, Javier Luis Cánovas Izquierdo, and Jesús Sánchez Cuadrado. "ModelSet: a dataset for machine learning in Model-driven Engineering." In: *Software and Systems Modeling* (2022), pp. 1–20. DOI: 10.1007/s10270-021-00929-3.

[210] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *An introduction to information retrieval*. Cambridge University Press, 2008. DOI: 10.1017/CBO9780511809071.

[211] Usman Mansoor, Marouane Kessentini, Philip Langer, Manuel Wimmer, Slim Bechikh, and Kalyanmoy Deb. "MOMM: Multi-objective model merging." In: *Journal of Systems and Software* 103 (2015), pp. 423–439. DOI: 10.1016/j.jss.2014.11.043.

[212] Stuart M Marshall, Cole Mathis, Emma Carrick, Graham Keenan, Geoffrey JT Cooper, Heather Graham, Matthew Craven, Piotr S Gromski, Douglas G Moore, Sara I Walker, et al. "Identifying molecules as biosignatures with assembly theory and mass spectrometry." In: *Nature communications* 12.1 (2021), p. 3033. DOI: 10.1038/s41467-021-23258-x.

[213] Jabier Martinez, Wesley K. G. Assunção, and Tewfik Ziadi. "ESPLA: A Catalog of Extractive SPL Adoption Case Studies." In: *Proceedings of the Software Product Line Conference (SPLC)*. ACM, 2017, pp. 38–41. DOI: 10.1145/3109729.3109748.

[214] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. "Bottom-up adoption of software product lines: a generic and extensible approach." In: *Proceedings of the Software Product Line Conference (SPLC)*. Ed. by Douglas C. Schmidt. ACM, 2015, pp. 101–110. DOI: 10.1145/2791060.2791086.

[215] Matias Martinez, Laurence Duchien, and Martin Monperrus. "Automatically Extracting Instances of Code Change Patterns with AST Analysis." In: *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2013, pp. 388–391. DOI: 10.1109/ICSM.2013.54.

[216] Víctor Martínez, Fernando Berzal, and Juan-Carlos Cubero. "A survey of link prediction in complex networks." In: *ACM computing surveys (CSUR)* 49.4 (2016), pp. 1–33.

[217] Steffen Mazanek and Mark Minas. "Business Process Models as a Showcase for Syntax-Based Assistance in Diagram Editors." In: *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Ed. by Andy Schürr and Bran Selic. Springer, 2009, pp. 322–336. DOI: 10.1007/978-3-642-04425-0_24.

[218] Steffen Mazanek and Mark Minas. "Generating correctness-preserving editing operations for diagram editors." In: *Electronic Communication of the European Association of Software Science and Technology* 18 (2009).

[219] Brendan D. McKay and Adolfo Piperno. "Practical graph isomorphism, II." In: *Journal of Symbolic Computation* 60 (2014), pp. 94–112. DOI: 10.1016/j.jsc.2013.09.003.

[220] Tom Mens. "Evolving Software Ecosystems A Historical and Ecological Perspective." In: *Dependable Software Systems Engineering* 40 (2015), pp. 170–192.

[221] Tom Mens and Serge Demeyer. *Software Evolution*. Springer, 2008.

[222] Tom Mens, Gabriele Taentzer, and Dirk Müller. "Challenges in model refactoring." In: *Proc. 1st Workshop on Refactoring Tools, University of Berlin*. Vol. 98. 2007, pp. 1–5.

[223] Tom Mens, Gabriele Taentzer, and Olga Runge. "Detecting structural refactoring conflicts using critical pair analysis." In: *Electronic Notes in Theoretical Computer Science* 127.3 (2005), pp. 113–128.

[224] Tom Mens and Pieter Van Gorp. "A taxonomy of model transformation." In: *Electronic Notes in Theoretical Computer Science* 152.1-2 (2006), pp. 125–142. DOI: 10.1016/j.entcs.2005.10.021.

[225] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. "Challenges in software evolution." In: *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*. IEEE. 2005, pp. 13–22. DOI: 10.1109/IWPSE.2005.7.

[226] Andreas Metzger and Klaus Pohl. "Software product line engineering and variability management: achievements and challenges." In: *Proceedings of the Conference on Future of Software Engineering (FOSE)*. ACM, 2014, pp. 70–84. DOI: 10.1145/2593882.2593888.

[227] Bart Meyers and Hans Vangheluwe. "A framework for evolution of modeling languages." In: *Science of Computer Programming* 76.12 (2011), pp. 1223–1246. DOI: 10.1016/j.scico.2011.01.002.

[228] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. "Distributed representations of words and phrases and their compositionality." In: *Advances in neural information processing systems* 26 (2013).

[229] John H Milsum. "The technosphere, the biosphere, the sociosphere: Their systems modeling and optimization." In: *IEEE Spectrum* 5.6 (1968), pp. 76–82. DOI: 10.1109/MSPEC.1968.5214690.

[230] Alok Mishra and Yehia Ibrahim Alzoubi. "Structured software development versus agile software development: a comparative analysis." In: *International Journal of System Assurance Engineering and Management* 14 (2023), pp. 1504–1522. DOI: 10.1007/s13198-023-01958-5.

[231] *Model-driven Engineerin (MDE)*. https://cio-wiki.org//wiki/Model_Driven_Engineering_(MDE). Accessed: 2024-05-09.

[232] *Model-driven engineering*. https://en.wikipedia.org/wiki/Model-driven_engineering. Accessed: 2024-04-28.

[233] Parastoo Mohagheghi and Jan Aagedal. "Evaluating Quality in Model-driven Engineering." In: *International Workshop on Modeling in Software Engineering (MISE'07: ICSE Workshop 2007)*. 2007, pp. 6–6. DOI: 10.1109/MISE.2007.6.

[234] Chihab eddine Mokaddem, Houari Sahraoui, and Eugene Syriani. "Recommending model refactoring rules from refactoring examples." In: *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*. ACM, 2018, pp. 257–266. DOI: 10.1145/3239372.3239406.

[235] Gunter Mussbacher et al. "Opportunities in Intelligent Modeling Assistance." In: *Software and Systems Modeling* 19.5 (Sept. 2020), 1045–1053. DOI: 10.1007/s10270-020-00814-5.

[236] Patrick Mäder, Tobias Kuschke, and Mario Janke. "Reactive Auto-Completion of Modeling Activities." In: *Transactions on Software Engineering* 47.7 (2021), pp. 1431–1451. DOI: 10.1109/TSE.2019.2924886.

[237] Nebras Nassar, Hendrik Radke, and Thorsten Arendt. "Rule-based repair of EMF models: An automated interactive approach." In: *Proceedings of the International Conference on Model Transformations (ICMT): Companion Proceedings*. Springer. 2017, pp. 171–181. DOI: 10.1007/978-3-319-61473-1_12.

[238] P Naur and B Randell. *Report on a conference sponsored by the NATO SCIENCE COMMITTEE Garmisch, Germany*. 1968.

[239] James Milne Neighbors. "Software construction using components." PhD thesis. University of California, Irvine, 1980.

[240] Patrick Neubauer, Robert Bill, Tanja Mayerhofer, and Manuel Wimmer. "Automated generation of consistency-achieving model editors." In: *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2017, pp. 127–137. DOI: 10.1109/SANER.2017.7884615.

[241] Hoan Anh Nguyen, Tien N. Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. "Graph-based mining of in-the-wild, fine-grained, semantic code change patterns." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM/IEEE, 2019, pp. 819–830. DOI: 10.1109/ICSE.2019.00089.

[242] Siegfried Nijssen and Joost N. Kok. "The Gaston tool for frequent subgraph mining." In: *Electronic Notes in Theoretical Computer Science* 127.1 (2005), pp. 77–87. DOI: 10.1016/j.entcs.2004.12.039.

[243] Manuel Ohrndorf, Christopher Pietsch, Udo Kelter, Lars Grunske, and Timo Kehrer. "History-Based Model Repair Recommendations." In: *Transactions of Software Engineering Methodology* 30.2 (2021). DOI: 10.1145/3419017.

[244] Manuel Ohrndorf, Christopher Pietsch, Udo Kelter, and Timo Kehrer. "ReVision: A tool for history-based model repair recommendations." In: *Proceedings of the International Conference on Software Engineering (ICSE): Companion Proceedings*. ACM, 2018, pp. 105–108. DOI: 10.1145/3183440.3183498.

[245] OMG. *OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1*. Tech. rep. Object Management Group, June 2013. URL: http://www.omg.org/spec/MOF/2.4.1.

[246] OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.3*. Tech. rep. Object Management Group, June 2016. URL: https://www.omg.org/spec/QVT.

[247] OMG. *Unified Modeling Language (UML) Version 2.5.1*. Standard. Object Management Group, Dec. 2017. URL: https://www.omg.org/spec/UML/2.5.1.

[248] OMG. *OMG SysML v. 1.6*. Standard. Object Management Group, Dec. 2019. URL: https://sysml.org/.res/docs/specs/OMGSysML-v1.6-19-11-01.pdf.

[249] Murat Oruc, Fuat Akal, and Hayri Sever. "Detecting Design Patterns in Object-Oriented Design Models by Using a Graph Mining Approach." In: *Proceedings of the International Conference in Software Engineering Research and Innovation (CONISOFT)*. IEEE, 2016, pp. 115–121. DOI: 10.1109/CONISOFT.2016.26.

[250] Long Ouyang et al. "Training language models to follow instructions with human feedback." In: *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*. 2022.

[251] David Lorge Parnas. "Software Aging." In: *Proceedings of the International Conference on Software Engineering*. ICSE '94. Washington, DC, USA: IEEE Computer Society Press, 1994, 279–287. DOI: 10.5555/257734.257788.

[252] David Lorge Parnas and Paul C Clements. "A rational design process: How and why to fake it." In: *IEEE transactions on software engineering* 2 (1986), pp. 251–257.

[253] Carlota Perez. "Technological revolutions, paradigm shifts and socio-institutional change." In: *Globalization, economic development and inequality: An alternative perspective* (2004), pp. 217–242.

[254] Tristan Pfofe, Thomas Thüm, Sandro Schulze, and Ina Schaefer. "Synchronizing software variants with VariantSync." In: *Proceedings of the Software Product Line Conference (SPLC)*. ACM, 2016, pp. 329–332. DOI: 10.1145/2934466.2962726.

[255] John Robinson Pierce. *An introduction to information theory: symbols, signals & noise*. Courier Corporation, 1980.

[256] Christopher Pietsch, Timo Kehrer, Udo Kelter, Dennis Reuling, and Manuel Ohrndorf. "SiPL–A Delta-Based Modeling Framework for Software Product Line Engineering." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE. IEEE, 2015, pp. 852–857. DOI: 10.1109/ASE.2015.106.

[257] Christopher Pietsch, Udo Kelter, Timo Kehrer, and Christoph Seidl. "Formal foundations for analyzing and refactoring delta-oriented model-based software product lines." In: *Proceedings of the Software Product Line Conference (SPLC)*. 2019, pp. 207–217. DOI: 10.1145/3336294.3336299.

[258] Pit Pietsch, Hamed Shariat Yazdi, and Udo Kelter. "Generating realistic test models for model processing tools." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2011, pp. 620–623. DOI: 10.1109/ASE.2011.6100140.

[259] Klaus Pohl, Günter Böckle, and Frank Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Jan. 2005.

[260] Michael Polanyi. *Personal Knowledge: Towards a Post Critical Philosophy*. University of Chicago Press, 1958, p. 428. DOI: 10.7208/chicago/9780226232768.001.0001.

[261] Oleksandr Polozov and Sumit Gulwani. "FlashMeta: A Framework for Inductive Program Synthesis." In: *SIGPLAN Not.* 50.10 (2015), 107–126. DOI: 10.1145/2858965.2814310.

[262] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. "Language models are unsupervised multitask learners." In: *OpenAI blog* 1.8 (2019), p. 9.

[263] Jan Ramon and Thomas Gärtner. "Expressivity versus efficiency of graph kernels." In: *Proceedings of the International Workshop on Mining Graphs, Trees and Sequences*. 2003, pp. 65–74.

[264] Nils Reimers and Iryna Gurevych. "Sentence-bert: Sentence embeddings using siamese bert-networks." In: *arXiv* (2019). DOI: 10.48550/arXiv.1908.10084.

[265] Pedro Ribeiro, Pedro Paredes, Miguel EP Silva, David Aparicio, and Fernando Silva. "A survey on subgraph counting: concepts, algorithms, and applications to network motifs and graphlets." In: *ACM Computing Surveys (CSUR)* 54.2 (2021), pp. 1–36.

[266] Jorma Rissanen. "Modeling by shortest data description." In: *Automatica* 14.5 (1978), pp. 465–471. DOI: 10.1016/0005-1098(78)90005-5.

[267] Adi Robertson. "The history of software." In: *IEEE Spectrum* 44.10 (2007), pp. 54–59.

[268] Gregorio Robles, Michel RV Chaudron, Rodi Jolak, and Regina Hebig. "A reflection on the impact of model mining from GitHub." In: *Information and Software Technology* 164 (2023), p. 107317. DOI: 10.1016/j.infsof.2023.107317.

[269] Alberto Rodrigues Da Silva. "Model-driven engineering: A survey supported by the unified conceptual model." In: *Computer Languages, Systems and Structures* 43 (2015), pp. 139–155. DOI: 10.1016/j.cl.2015.06.001.

[270] Louis M. Rose et al. "Graph and model transformation tools for model migration: Empirical results from the transformation tool contest." In: *Software & Systems Modeling* 13.1 (2014), pp. 323–359. DOI: 10.1007/s10270-012-0245-0.

[271] Louis Mathew Rose. "Structures and Processes for Managing Model-Metamodel Co-evolution." In: *Evolution* July (2011).

[272] Winston W. Royce. "Managing the Development of Large Software Systems." In: (1970), pp. 1–9. DOI: 10.7551/mitpress/12274.003.0035.

[273] Julia Rubin and Marsha Chechik. "N-way model merging." In: *Proceedings of the Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2013, pp. 301–311. DOI: 10.1145/2491411.2491446.

[274] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. "Managing cloned variants: A framework and experience." In: *Proceedings of the Software Product Line Conference (SPLC)*. ACM, 2013, pp. 101–110. DOI: 10.1145/2491627.2491644.

[275] Julia Rubin, Andrei Kirshin, Goetz Botterweck, and Marsha Chechik. "Managing forked product variants." In: *Proceedings of the Software Product Line Conference (SPLC)*. Vol. 1. 2012, pp. 156–160. DOI: 10.1145/2362536.2362558.

[276] Uwe Ryssel. "Automatische Generierung von feature-orientierten Produktlinien aus Varianten von funktionsblockorientierten Modelen." PhD thesis. Technische Universität Dresden, Nov. 2014, p. 246.

[277] Uwe Ryssel, Joern Ploennigs, and Klaus Kabitzsch. "Extraction of feature models from formal contexts." In: Aug. 2011, p. 4. DOI: 10.1145/2019136.2019141.

[278] Hajer Saada, Marianne Huchard, Michel Liquiere, and Clémentine Nebut. "Learning model transformation patterns using graph generalization." In: *Proceedings of the International Conference on Concept Lattices and Their Applications*. CEUR-WS.org, 2014, pp. 11–22.

[279] Hazem Peter Samoaa, Firas Bayram, Pasquale Salza, and Philipp Leitner. "A systematic mapping study of source code representation for deep learning in software engineering." In: *IET Software* (2022). DOI: 10.1049/sfw2.12064.

[280] Maxime Savary-Leblanc, Lola Burgueño, Jordi Cabot, Xavier Le Pallec, and Sébastien Gérard. "Software assistants in software engineering: A systematic mapping study." In: *Software: Practice and Experience* 53.3 (2023), pp. 856–892. DOI: 10.1002/spe.3170.

[281] Walter Scacchi. "Models of Software Evolution: Life Cycle and Process." In: *Software Engineering Institute* (2019). DOI: 10.1184/R1/6575687.v1.

[282] Ina Schaefer. "Variability Modeling for Model-driven Development of Software Product Lines." In: *VaMoS* 10 (2010), pp. 85–92.

[283] D.C. Schmidt. "Guest Editor's Introduction: Model-driven Engineering." In: *Computer* 39.2 (2006), pp. 25–31. DOI: 10.1109/MC.2006.58.

[284] Maik Schmidt and Tilman Gloetzner. "Constructing difference tools for models using the SiDiff framework." In: *Proceedings of the International Conference on Software Engineering (ICSE): Companion Proceedings*. ACM/IEEE, 2008, pp. 947–948. DOI: 10.1145/1370175.1370201.

[285] Maik Schmidt, Sven Wenzel, Timo Kehrer, and Udo Kelter. "History-based merging of models." In: *ICSE Workshop on Comparison and Versioning of Software Models (CVSM)*. IEEE. 2009, pp. 13–18. DOI: 10.1109/CVSM.2009.5071716.

[286] Thomas Schmorleiz and Ralf Lämmel. "Similarity management via history annotation." In: *SATToSE 2014—Pre-proceedings* (2014), p. 45.

[287] B Schölkopf. *Learning with kernels: support vector machines, regularization, optimization, and beyond*. The MIT Press, 2002. DOI: 10.7551/mitpress/4175.001.0001.

[288] Gunnar Schröder, Maik Thiele, and Wolfgang Lehner. "Setting goals and choosing metrics for recommender system evaluations." In: *Proceedings of the Conference on Recommender Systems (RecSys)*. ACM, 2011, p. 53.

[289] Ken Schwaber. "The SCRUM Development Process." In: (1995), pp. 117–134. DOI: 10.1007/978-1-4471-0947-1_11.

[290] Felix Schwägerl. "Version Control and Product Lines in Model-driven Software Engineering." dissertation. 2018.

[291] Lukas Selvaggio. "Feature Identification with Formal Concept Analysis: A Case Study." MA thesis. Saarland Informatics Campus, Saarland University, 2022. URL: https://www.se.cs.uni-saarland.de/theses/LukasSelvaggioBA.pdf.

[292] Sagar Sen, Benoit Baudry, and Hans Vangheluwe. "Towards Domain-specific Model Editors with Automatic Model Completion." In: *Simulation* 86.2 (2010), pp. 109–126. DOI: 10.1177/0037549709340530.

[293] A-D Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, et al. "Mining features from the object-oriented source code of software variants by combining lexical and structural similarity." In: *2013 IEEE 14th International Conference on Information Reuse & Integration (IRI)*. IEEE. 2013, pp. 586–593.

[294] Hamed Shariat Yazdi, Lefteris Angelis, Timo Kehrer, and Udo Kelter. "A framework for capturing, statistically modeling and analyzing the evolution of software models." In: *Journal of Systems and Software* 118 (2016), pp. 176–207. ISSN: 0164-1212. DOI: 10.1016/j.jss.2016.05.010.

[295] Abhishek Sharma, Dániel Czégel, Michael Lachmann, Christopher P Kempes, Sara I Walker, and Leroy Cronin. "Assembly Theory Explains and Quantifies the Emergence of Selection and Evolution." In: *arXiv preprint* (2022). DOI: 10.48550/arXiv.2206.02279.

[296] Steven She. "Feature model synthesis." PhD thesis. 2013.

[297] Steven She, Uwe Ryssel, Nele Andersen, Andrzej Wasowski, and Krzysztof Czarnecki. "Efficient synthesis of feature models." In: *Inf. Softw. Technol.* 56.9 (2014), pp. 1122–1143. DOI: 10.1016/J.INFSOF.2014.01.012.

[298] Michael Shermer. "Patternicity: Finding meaningful patterns in meaningless noise." In: *Scientific American* 299.5 (2008), p. 48.

[299] Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. "Weisfeiler-lehman graph kernels." In: *Journal of Machine Learning Research* 12.9 (2011).

[300] Yunsheng Shi, Zhengjie Huang, Shikun Feng, Hui Zhong, Wenjin Wang, and Yu Sun. "Masked label prediction: Unified message passing model for semi-supervised classification." In: *arXiv preprint* (2020).

[301] Sohil Lal Shrestha and Christoph Csallner. "SLGPT: using transfer learning to directly generate simulink model files and find bugs in the simulink toolchain." In: *Proceedings of the Conference on Evaluation and Assessment in Software Engineering (EASE)*. ACM, 2021. DOI: 10.1145/3463274.3463806.

[302] Janet Siegmund, Norbert Siegmund, and Sven Apel. "Views on internal and external validity in empirical software engineering." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE. 2015, pp. 9–19. DOI: 10.1109/ICSE.2015.24.

[303] Herbert A. Simon. *Models of Bounded Rationality: Vol. 1, Economic Analysis and Public Policy*. MIT Press, 1982.

[304] Martin Simonovsky and Nikos Komodakis. "Graphvae: Towards generation of small graphs using variational autoencoders." In: *International conference on artificial neural networks*. Springer. 2018, pp. 412–422.

[305] Dominik Sobania, Martin Briesch, and Franz Rothlauf. "Choose Your Programming Copilot: A Comparison of the Program Synthesis Performance of GitHub Copilot and Genetic Programming." In: *arXiv* (2021). DOI: 10.48550/arXiv.2111.07875.

[306] Jascha Sohl-Dickstein, Eric Weiss, Niru Maheswaranathan, and Surya Ganguli. "Deep unsupervised learning using nonequilibrium thermodynamics." In: *International conference on machine learning*. PMLR. 2015, pp. 2256–2265.

[307] Thomas Stahl, Markus Völter, and Krzysztof Czarnecki. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, Inc., 2006. DOI: 10.5555/1196766.

[308] Stefan Stănciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wąsowski. "Concepts, operations, and feasibility of a projection-based variation control system." In: *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2016, pp. 323–333. DOI: 10.1109/ICSME.2016.88.

[309] Friedrich Steimann and Bastian Ulke. "Generic Model Assist." In: *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Ed. by Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter Clarke. Springer Berlin Heidelberg, 2013, pp. 18–34. DOI: 10.1007/978-3-642-41533-3_2.

[310] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[311] Matthew Stephan. "Towards a cognizant virtual software modeling assistant using model clones." In: *Proceedings of the International Conference on Software Engineering (ICSE) (NIER)*. Ed. by Anita Sarma and Leonardo Murta. IEEE / ACM, 2019, pp. 21–24. DOI: 10.1109/ICSE-NIER.2019.00014.

[312] Matthew Stephan. "Towards a cognizant virtual software modeling assistant using model clones." In: *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE. 2019, pp. 21–24.

[313] Matthew Stephan and James R Cordy. "A Survey of model comparison approaches and applications." In: *Proceedings of the International Conference on Model-driven Engineering and Software Development (MODELSWARD)*. 2013, pp. 265–277. DOI: 10.5220/0004311102650277.

[314] Linda Studer, Jannis Wallau, Rolf Ingold, and Andreas Fischer. "Effects of graph pooling layers on classification with graph neural networks." In: *2020 7th Swiss Conference on Data Science (SDS)*. IEEE. 2020, pp. 57–58.

[315] Yu Sun, Jeff Gray, and Jules White. "MT-Scribe: An end-user approach to automate software model evolution." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM/IEEE, 2011, pp. 980–982. ISBN: 9781450304450. DOI: 10.1145/1985793.1985966.

[316] Vinitra Swamy, Angelika Romanou, and Martin Jaggi. "Interpreting Language Models Through Knowledge Graph Extraction." In: *arXiv* (2021). DOI: 10.48550/arXiv.2111.08546.

[317] Gabriele Taentzer, André Crema, René Schmutzler, and Claudia Ermel. "Generating domain-specific model editors with complex editing commands." In: *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*. Springer. 2007, pp. 98–103. DOI: 10.1007/978-3-540-89020-1_8.

[318] Gabriele Taentzer, Manuel Ohrndorf, Yngve Lamo, and Adrian Rutle. "Change-Preserving Model Repair." In: *Fundamental Approaches to Software Engineering*. Ed. by Marieke Huisman and Julia Rubin. Springer Berlin Heidelberg, 2017, pp. 283–299. DOI: 10.1007/978-3-662-54494-5_16.

[319] *The Fast Guide to Model Driven Architecture*. https://www.omg.org/mda/mda_files/Cephas_MDA_Fast_Guide.pdf. Accessed: 2023-04-18.

[320] Christof Tinnes. "Improving Trace Link Recommendation by Using Non-Isotropic Distances and Combinations." In: *arXiv preprint* (2023). DOI: 10.48550/arXiv.2307.07781.

[321] Christof Tinnes, Thomas Fuchß, Uwe Hohenstein, and Sven Apel. "Towards Automatic Support of Software Model Evolution with Large Language˜ Models." In: *arXiv preprint* (2023). DOI: 10.48550/arXiv.2312.12404.

[322] Christof Tinnes, Uwe Hohenstein, Wolfgang Rössler, and Andreas Biesdorf. "Tackling Model Drifts in Industrial Model-driven Software Product Lines by Means of a Graph Database." In: *Proceedings of the International Conference on Data Science, Technology and Applications, DATA*. SCITEPRESS, 2022. DOI: 10.5220/0011319800003269. URL: https://doi.org/10.5220/0011319800003269.

[323] Christof Tinnes, Timo Kehrer, Mitchell Joblin, Uwe Hohenstein, Andreas Biesdorf, and Sven Apel. "Mining domain-specific edit operations from model repositories with applications to semantic lifting of model differences and change profiling." In: *Automated Software Engineering* 30.2 (2023), p. 17. DOI: 10.1007/s10515-023-00381-1.

[324] Christof Tinnes, Timo Kehrer, Joblin. Mitchell, Uwe Hohenstein, Andreas Biesdorf, and Sven Apel. "Learning domain-specific edit operations from model repositories with frequent subgraph mining." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM/IEEE, 2021. DOI: 10.1109/ASE51524.2021.9678698.

[325] Christof Tinnes, Wolfgang Rössler, Uwe Hohenstein, Torsten Kühn, Andreas Biesdorf, and Sven Apel. "Sometimes you have to treat the symptoms: tackling model drift in an industrial clone-and-own software product line." In: *Proceedings of the Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2022, pp. 1355–1366. DOI: 10.1145/3540250.3558960.

[326] Christof Tinnes, Alisa Welter, and Sven Apel. "Software Model Evolution with Large Language Models: Experiments on Simulated, Public, and Industrial Datasets." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2025.

[327] Alan Mathison Turing. "On computable numbers, with an application to the Entscheidungsproblem." In: *Proceedings of the London mathematical society* 2.1 (1936), pp. 230–265.

[328] Arie Van Deursen, Eelco Visser, and Jos Warmer. "Model-driven software evolution: A research agenda." In: *Technical Report Series TUD-SERG-2007-006.* (2007).

[329] Dániel Varró. "Model transformation by example." In: *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Springer, 2006, pp. 410–424. DOI: 10.1007/11880240_29.

[330] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need." In: *Advances in Neural Information Processing Systems* 30 (2017). DOI: 10.5555/3295222.3295349.

[331] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. "Graph attention networks." In: *arXiv preprint* (2017).

[332] Nikolai K Vereshchagin and Paul MB Vitányi. "Kolmogorov's structure functions and model selection." In: *IEEE Transactions on Information Theory* 50.12 (2004), pp. 3265–3290. DOI: 10.1109/TIT.2004.838346.

[333] Pieter E. Vermaas, Peter Kroes, Ibo van de Poel, Maarten Franssen, and Wybo Houkes. *A Philosophy of Technology: From Technical Artefacts to Sociotechnical Systems*. Synthesis Lectures on Engineers, Technology and Society. Morgan & Claypool Publishers, 2011. DOI: 10.2200/S00321ED1V01Y201012ETS014.

[334] S Vichy N Vishwanathan, Nicol N Schraudolph, Risi Kondor, and Karsten M Borgwardt. "Graph kernels." In: *Journal of Machine Learning Research* 11 (2010), pp. 1201–1242. DOI: 1756006.1859891.

[335] Birgit Vogel-Heuser, Christoph Legat, Jens Folmer, and Stefan Feldmann. *Researching evolution in industrial plant automation: Scenarios and documentation of the pick and place unit*. Tech. rep. Institute of Automation and Information Systems, Technische Universität München, 2014.

[336] Christopher S Wallace. *Statistical and inductive inference by minimum message length*. Springer Science & Business Media, 2005.

[337] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. "What Do They Capture? - A Structural Analysis of Pre-Trained Language Models for Source Code." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2022, pp. 2377–2388. DOI: 10.1145/3510003.3510050.

[338] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R Lyu. "No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence." In: *Proceedings of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, pp. 382–394. DOI: 10.1145/3540250.3549113.

[339] Guan Wang, Francois Bernard Lauze, and Aasa Feragen. "Graph2graph learning with conditional autoregressive models." In: *arXiv preprint* (2021). DOI: 10.48550/arXiv.2106.03236.

[340] Pascal Welke, Tamás Horváth, and Stefan Wrobel. "Probabilistic and exact frequent subtree mining in graphs beyond forests." In: *Machine Learning* 108.7 (2019). DOI: 10.1007/s10994-019-05779-1.

[341] Pascal Welke, Florian Seiffarth, Michael Kamp, and Stefan Wrobel. "HOPS: Probabilistic subtree mining for small and large graphs." In: *Proceedings of the Conference on Knowledge Discovery (KDD)*. ACM, 2020, pp. 1275–1284. DOI: 10.1145/3394486.3403180.

[342] Patrick Welke, Nils Kißig, and Guido Moerkotte. "Efficient Frequent Subgraph Mining in Graphs of Bounded Tree-width." In: *Proceedings of the International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE. 2020.

[343] Alisa Welter. "Towards Model Editing Pattern Detection via Graph Variational Autoencoders." MA thesis. Saarland Informatics Campus, Saarland University, 2023.

[344] Nathan Weston, Ruzanna Chitchyan, and Awais Rashid. "A framework for constructing semantically composable feature models from natural language requirements." In: *Proceedings of the Software Product Line Conference (SPLC)*. 2009, pp. 211–220. DOI: 10.5555/1753235.1753265.

[345] Martin Weyssow, Houari Sahraoui, and Eugene Syriani. "Recommending metamodel concepts during modeling activities with pre-trained language models." In: *Software and Systems Modeling* 21.3 (2022), pp. 1071–1089. DOI: 10.1007/s10270-022-00975-5.

[346] *What is model-based systems engineering (MBSE)?* https://www.ibm.com/topics/model-based-systems-engineering. Accessed: 2023-04-18.

[347] Jon Whittle and et al. *Executable UML: A Foundation for Model-driven Architecture*. Addison-Wesley, 2002. DOI: 10.5555/545976.

[348] Niklaus Wirth. "A brief history of software engineering." In: *IEEE Annals of the History of Computing* 30.3 (2008).

[349] Claes Wohlin, Martin Höst, and Kennet Henningsson. "Empirical research methods in web and software engineering." In: *Web engineering* (2006), pp. 409–430. DOI: 10.1007/3-540-28218-1_13.

[350] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. "A comprehensive survey on graph neural networks." In: *Transactions on neural networks and learning systems* 32.1 (2020). DOI: 10.1109/TNNLS.2020.2978386.

[351] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. "A Systematic Evaluation of Large Language Models of Code." In: *Proceedings of the International Symposium on Machine Programming*. ACM, 2022, 1–10. DOI: 10.1145/3520312.3534862.

[352] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. "A systematic evaluation of large language models of code." In: *Proceedings of the International Symposium on Machine Programming*. 2022, pp. 1–10.

[353] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. "How powerful are graph neural networks?" In: *arXiv preprint* (2018). DOI: 10.48550/arXiv.1810.00826.

[354] Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. "What can neural networks reason about?" In: *arXiv preprint* (2019).

[355] Xifeng Yan and Jiawei Han. "gSpan: Graph-Based Substructure Pattern Mining." In: 3 (2002).

[356] Xifeng Yan and Jiawei Han. "Closegraph: mining closed frequent graph patterns." In: *Proceedings of the Conference on Knowledge Discovery (KDD)*. 2003, pp. 286–295. DOI: 10.1145/956750.956784.

[357] Junhan Yang, Zheng Liu, Shitao Xiao, Chaozhuo Li, Defu Lian, Sanjay Agrawal, Amit Singh, Guangzhong Sun, and Xing Xie. "Graphformers: Gnn-nested transformers for representation learning on textual graph." In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 28798–28810.

[358] Kentaro Yoshimura, Dharmalingam Ganesan, and Dirk Muthig. "Assessing merge potential of existing engine control systems into a product line." In: *International Workshop on Software Engineering for Automotive Systems*. 2006, pp. 61–67. DOI: 10.1145/1138474.1138485.

[359] Yijun Yu, Thein Than Tun, and Bashar Nuseibeh. "Specifying and detecting meaningful changes in programs." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2011, pp. 273–282. DOI: 10.1109/ASE.2011.6100063.

[360] Laura Zager. "Graph similarity and matching." PhD thesis. Massachusetts Institute of Technology, 2005.

[361] Liping Zhao, Waad Alhoshan, Alessio Ferrari, Keletso J Letsholo, Muideen A Ajagbe, Erol-Valeriu Chioasca, and Riza T Batista-Navarro. "Natural language processing for requirements engineering: a systematic mapping study." In: *ACM Computing Surveys (CSUR)* 54.3 (2021), pp. 1–41. DOI: 10.1145/3444689.

[362] Alice Zheng. *Evaluating machine learning models: a beginner's guide to key concepts and pitfalls*. O'Reilly Media, 2015.

[363] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. "What the fork: A study of inefficient and efficient forking practices in social coding." In: *Proceedings of the Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Tallinn, Estonia: ACM, 2019, 350–361. DOI: 10.1145/3338906.3338918.

[364] Rustam Zhumagambetov, Ferdinand Molnár, Vsevolod A Peshkov, and Siamac Fazli. "Transmol: repurposing a language model for molecular generation." In: *RSC advances* 11.42 (2021), pp. 25921–25932. DOI: 10.1039/D1RA03086H.