

**TECHNISCHE  
UNIVERSITÄT  
DRESDEN**

---

Fakultät Informatik Institut für Software- und Multimediatechnik, Professur für Softwaretechnologie

---

Student Research Project

# Synthetic History Graph Generation for Model-Driven Software Engineering

Leonard Sonnenberg

Born on: 09.09.1997 in Berlin

27.03.2026

Supervisor

M.Sc. Karl Kegel, Dr.-Ing. Sebastian Götz

Supervising professor

Prof. Dr. Uwe Aßmann

### Statement of authorship

I hereby certify that I have authored this document entitled *Synthetic History Graph Generation for Model-Driven Software Engineering* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, 27.03.2026

Leonard Sonnenberg

## **Abstract**

Synthetic data generation is becoming more and more important, not only for testing and benchmarking but also for training artificial intelligence models. In the context of model-driven software engineering, you need good model data to properly evaluate and build tools. While various approaches exist for generating model data that conforms to specified metamodels, the problem of generating evolving model data still needs exploration. In this work, we introduce an algorithm that uses the incremental nature of some graph generators to create a branched history graph. The resulting history graph is acyclic and has incremental changes between successive commits, simulating realistic model evolution. To support flexibility and extensibility, we design an adapter pattern that allows us to exchange the underlying model generator during the configuration of the history generation process. We evaluate the computational overhead of our algorithm by comparing its runtime to that of the underlying model generators without history construction.

# Contents

Abstract	3
<b>1 Introduction</b>	<b>6</b>
1.1 Motivation	6
1.2 Problem Space	6
1.3 Approach	7
1.4 Research Questions	8
1.5 Method and Contribution	9
<b>2 Background</b>	<b>10</b>
2.1 model-driven Software Engineering	10
2.1.1 Models and Metamodels	10
2.1.2 Model transformation	12
2.1.3 Model evolution	12
2.2 Synthetic Data Generation	14
2.2.1 Test Data Generation	14
2.2.2 Generative Models	15
2.3 Version Control Systems	16
2.4 Graph Generators	19
2.4.1 Traditional Graph Generators	19
2.4.2 Deep Graph Generators	19
2.4.3 Graph modification	20
2.4.4 Open Challenges	20
<b>3 Related Works</b>	<b>21</b>
3.1 The Need for MDSE Model Data	21
3.2 MDSE Model Data Collection	22
3.3 Graph Generation	22
<b>4 Conceptualization</b>	<b>24</b>
4.1 Requirements	25
4.2 History	26
4.3 Adaptability	27
<b>5 Implementation</b>	<b>29</b>
5.1 Abstract Algorithm	29
5.1.1 History Generation	30

5.1.2	History Population . . . . .	30
5.1.3	History Export . . . . .	31
5.2	Branches . . . . .	31
5.2.1	Spawning New Branches . . . . .	31
5.2.2	Branch variables . . . . .	32
5.3	Random . . . . .	33
5.3.1	The Random Sequence . . . . .	33
5.4	Graph Generation . . . . .	34
5.4.1	GraphGentool . . . . .	34
5.4.2	Circle Generator . . . . .	35
5.5	Merge Generation . . . . .	36
5.6	Default Values . . . . .	37
<b>6</b>	<b>Evaluation</b>	<b>39</b>
6.1	History generation . . . . .	39
6.2	History population . . . . .	42
6.2.1	Circle Graph Generator . . . . .	43
6.2.2	GraphGentool . . . . .	45
6.2.3	Merges . . . . .	46
6.3	Research Questions . . . . .	47
<b>7</b>	<b>Conclusion</b>	<b>49</b>

# 1 Introduction

## 1.1 Motivation

The reliability and performance of software systems depend on thorough testing and benchmarking. Both approaches require realistic data to have meaningful results. There are two main ways of getting data for those tests and benchmarks. The first is to source it from one or more live systems. The second is to generate synthetic data using a dedicated generator. Data gathered from live systems is generally closer to the expected deployment-time data. But it is usually more difficult to obtain, because companies do not share their proprietary models. Real data can also suffer from quality issues, like syntactically invalid models and semantically incorrect structures [Ver+25]. Synthetic data, on the other hand, can be produced in abundance once a suitable generator is developed. Additionally, it is often possible to configure the generation process to give the generated data some desired properties. This can be very useful when selecting data for tests and benchmarks.

In their paper "A Variance-Based Drift Metric for Inconsistency Estimation in Model Variant Sets." [Keg+24] Kegel et al. introduce a generator called GraphGentool. This generator is designed to create synthetic model graphs with different variants. As seen in Figure 4.1, these graphs consist of directed edges and nodes. Each node is either a simple node with a color or a region that contains another graph. GraphGentool generates the entire graph at once using a list of ratios and the size of the entire model as the total sum of nodes and edges. To introduce different variants, the final model graph is then altered using probabilistic changes. This results in a history graph as seen in Figure 1.1 We aim to improve upon the existing tool to generate a meaningful history for a model graph or a continuation of an existing model graph.

## 1.2 Problem Space

MDSE models are commonly represented by graphs, like UML class diagrams or state machines. Depending on the metamodel of a project, the structure of those graphs can vary widely across projects. Furthermore, each project has a development history containing different versions of the graph and the connections between them. Each version of the graph is internally consistent with the metamodel and part of a progression with incremental changes between neighboring versions. However, the history is not just linear. At any point there can be parallel versions that are developed simultaneously and are later merged or retired.

A limitation of the GraphGentool is that the variants it generates are all linear, as seen in Figure 1.1. The parameters used to generate those variants are the number of variants, the number of edits, the way of counting the edits, the locality of the edits, and the probabilities of

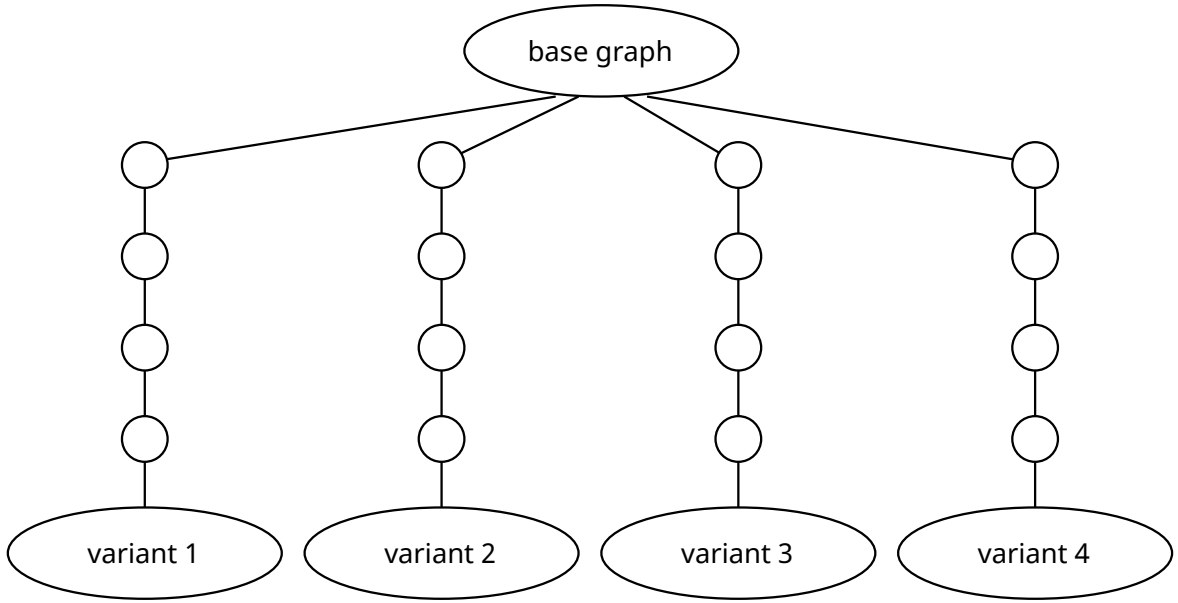


Figure 1.1: Example history graph of the GraphGentool. After generating the model graph, GraphGentool can generate variants. In this example four variants were generated, each with a distance of five atomic actions to the base graph.

each possible edit operation. For a sufficiently big model graph and comparatively low number of variants, it is unlikely that the first edit of two variants is the same. Therefore, variants will most likely all branch off the base graph without any common in-between versions. This means GraphGentool is not able to generate the branched history we aim to generate.

To the best of our knowledge, there are no graph generators that support generating a branched history with multiple versions progressing simultaneously. We also want to generate a broad range of MDSE model graphs. Therefore, we must be able to adapt the generator to very different constraints depending on the tools and metamodels of the MDSE project. At the same time, we need to ensure the integrity of the history by only allowing small changes between versions.

Additionally, we need to ensure that the generator stays deterministic. By allowing to pass a seed used by the random generator, GraphGentool ensures replayability, which is key for regression tests and debugging. We similarly want to ensure that we are able to replay the generation process as well.

### 1.3 Approach

There exists a variety of different graph generator algorithms that can generate graphs [Xia+22]. When designing our generator, we assume that the overall features of the history are independent from the topology of the MDSE model graph. This allows us to generate the history and the model graph independently from each other and therefore incorporate existing graph generators, like GraphGentool, that support incremental changes but do not support a branched history.

To do that we define the history as a set of commits linked by directed edges. Each commit holds an MDSE model graph and represents either a set of atomic edits on this graph or the merge of two graphs into one in the case of a merge commit. We can then generate the history as a graph of commits and populate it in a later step.

To make the MDSE model graph coherently progress from commit to commit, we approach

its generation progressively, similar to how it would progress in a real project. This means each commit depends on its predecessors to calculate its graph. This allows us to use any graph generator that can generate an amount of atomic edits on top of a base graph. Each commit uses the MDSE model graph of the previous commit as the base graph for the generation of its graph. This progressively populates the history, leaving only the issue of merge conflicts. Each merge only depends on the two previous MDSE model graphs. Since the history is filled progressively, both previous model graphs have been generated by the time the merge commit is filled. This allows us to use existing graph merge algorithms to calculate the merged graphs. Since merging is another problem space of its own, we will not implement our own merger. Instead we design an interface to easily exchange the merger.

### 1.4 Research Questions

In the following section we present the research questions we aim to answer.

#### **RQ1: Can we improve upon the existing generators to generate a realistic development history?**

In the survey by Zou et al. [Zou+19] from 2019, they filtered all projects on GitHub down to 2,923 projects that are representative of collaborative development using Git. We use the statistical data they gathered to find realistic values for our history. For the general size of the history graphs, they found that the projects had a mean of 918.3 commits with a maximum of 25,052 commits. For branches, on the other hand, 87.9% only had five or less, while only 3.1% had more than 30. We consider the question answered if we manage to generate a history using our algorithm that matches the statistical data of the GitHub repositories from the 2019 study. Additionally, we need to populate the history with model graph data using an existing generator for this question to be answered positively.

#### **RQ1a: Can we generate the branched history without an unreasonable increase in compute time compared to a linear one?**

Since we want to separate the generation of the history and the generation of the MDSE model graphs, we can measure the history generation time and history population time separately. We consider the question answered after we have multiple sample runs with different random seeds and history sizes that show the computational overhead of the history generation. The question is answered positively if the history generation time is at least two orders of magnitude lower than the history population time.

#### **RQ1b: What is the runtime to generate the MDSE model graphs for a realistic history?**

Using the statistical GitHub repository data from the 2019 study, we choose values for a realistic history size. We consider the question answered, once we have multiple sample runs on a consumer-grade workstation with different seeds and a realistic history size.

#### **RQ2: How does a modular design look like that allows to extend the history generation with different MDSE model graph generators?**

We consider the question answered, if we showcase that we can use multiple graph generators without altering the history generation, and we discussed the development effort to make those generators available to the history population.



## 1.5 Method and Contribution

This work is split into different sections. In the [background](#) section, we present fundamental concepts from the topics of [model-driven software engineering](#), [synthetic data generation](#), [version control systems](#), and [graph generators](#). The next section after that is the [related works](#) section. There we have a look at a few works that take on similar problems or work in similar technical spaces. After that comes the [conceptualization](#) section. There we break down the requirements and how we plan on achieving them. The next section is the [implementation](#) section. There we elaborate on our implementation and go into detail on how each part interacts with the others. The last section is the [evaluation](#) section. There we quantitatively measure the outputs and generation times of our implementation. This is also where we relate back to the research questions.

This work contributes an algorithm that allows the generation of a branched history and interfaces that make it possible for graph generators to fill it with incremental changes. A minor contribution is the circle graph generator that can generate graphs consisting of multiple interconnected circles.

## 2 Background

This section presents topics that are relevant for the work. These four topics are model-driven software engineering, synthetic data generation, version control systems, and graph generators.

### 2.1 model-driven Software Engineering

model-driven software engineering is an approach within software development that focuses on models instead of source code as the primary artifact of development.

#### 2.1.1 Models and Metamodels

Models are a collection of facts about a system under study [Sei03]. Each fact characterizes a feature or an attribute and can be either true or false in relation to that system. In conventional software development models are the specifications of the project. In MDSE we understand models as representations of reality, with each model having a specific objective. Pidd articulates in his book on modeling that *"a model is an external and explicit representation of part of reality as seen by the people who wish to use that model to understand, to change, to manage, and to control that part of reality"* [Pid09]. This highlights that each model is inherently reductive, representing only a specific view on reality. An example that Pidd uses is that of a car. Some people might see it as something that transports you from A to B, while others might see it as a status symbol or as a hobby.

MDSE focuses on the development of models and their formalization, enrichment, interdependence, and transformations. It is common for MDSE projects to contain multiple models. One example is seen in Figure 2.1. These focus on specific aspects of the system, like the domain model and business model, describing domain concepts and business requirements, respectively [AZW06]. But the models are not limited to requirements. They also contain architectural concerns, e.g., in the platform-independent model, and implementation concerns, e.g., in the platform-specific model.

Each model is expressed in a language, called the modeling language. However, this language can be described by another model. This model is called the metamodel. It defines the elements of the modeling language. This can be repeated as the metamodel is also expressed using a modeling language. The resulting hierarchy is called the meta-pyramid [AZW06]. This pyramid ends at the metametamodel level, as those models are self-describing and do not require a higher-level definition. A typical modeling language is UML. It is a general-purpose

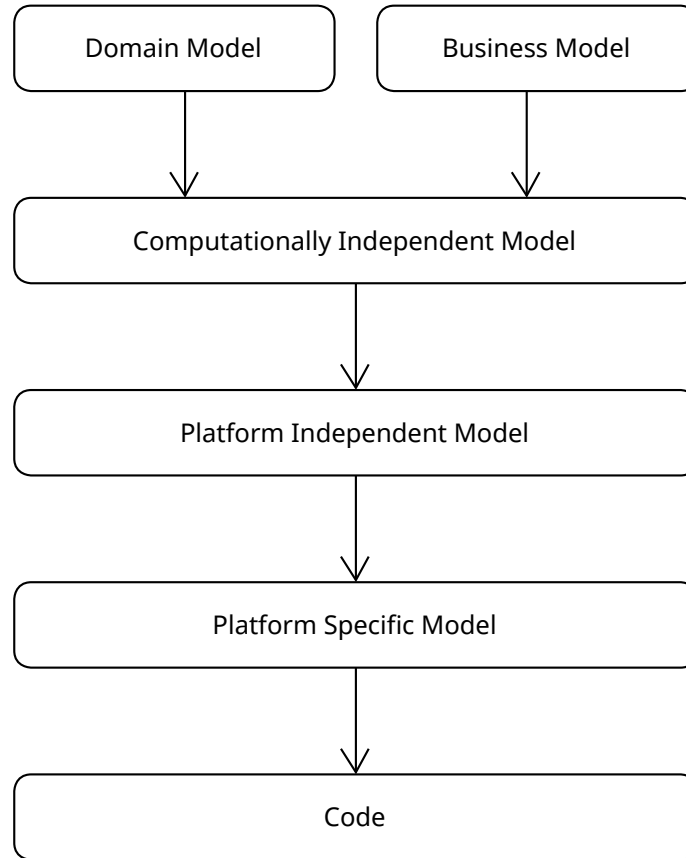


Figure 2.1: Simplified pipeline of a model translational framework: The domain model holds domain-specific knowledge, describing facts about and dependencies between domain-specific concepts. The business model describes the business concern and requirements. By collecting all requirements and relevant knowledge, one computationally independent model is created. From there a platform-independent model can be devised. It describes relevant entities and interactions between them without any platform-specific features. To create the platform-specific model, platform-specific extensions are added to the platform-independent model. The platform-specific model then contains all information needed to generate the code for the project. Using such a pipeline allows companies to easily develop multiple products in a product line by switching the domain model or bring their product to multiple platforms by switching the platform-specific extensions.

modeling language that can be used in different domains. There are also domain-specific modeling languages. They support development when working in a specific domain. To do that, they lift domain-specific concepts into the metamodel layer. This limits the resulting models to the specified domain but makes developing in those domains a lot easier [Sch+06]. Working with a domain-specific language does not limit a project to one domain. Different parts of a system can be described in a different modeling language.

For the purpose of this work, we consider MDSE models to be their graph representations. Since we do not want to generate models only for a specific domain, the metamodel of these graphs is very generic. The goal is to design the generation algorithm to be open and easily interchangeable so if there is a demand for more specific constraints on the models, each generator can be easily adjusted or exchanged.

### 2.1.2 Model transformation

As already seen in Figure 2.1, in MDSE we want to transform one model into another. There are a variety of different transformations that can transform a source model into a target model. The following paragraph summarizes the five dimensions Mens et al. [MV06] find in their taxonomy that differentiate model transformations based on the characteristics of the source and target model.

**Number of source and target models:** Model transformations do not have to be one-to-one, with one source model being transformed into one target model. For example, in the translational pipeline of Figure 2.1, the transformation from the platform-independent model to the platform-specific model can be extended to create multiple platform-specific models at once. This would then be a one-to-many transformation. Additionally, in Figure 2.1, we see that the domain model and the business model are joined into the computationally independent model. This is an example of a many-to-one transformation.

**Technical space:** A technical space is a framework that contains concepts, tools, techniques, and languages of a corresponding technology. When transforming a model from one technical space into a model from a different technical space, there needs to be a tool or toolchain bridging both technical spaces.

**Endogenous versus exogenous:** Models are expressed using a modeling language. If the modeling language of the source model is different from the one of the target model, the transformation is exogenous. If both models are in the same language, the transformation is endogenous. An example of exogenous transformations is compiling to JavaScript. Source code is also a model with the programming language as the modeling language. Therefore, when generating JavaScript from other object-oriented programming languages, the input and output model will have a different modeling language. For endogenous transformation, an example is refactoring. The model language stays the same, but the model itself is changed.

**Horizontal versus vertical:** Models can have different levels of abstraction. When the source and target models have a different level of abstraction, the transformation is called vertical. If they have the same level of abstraction, the transformation is called horizontal. When generating source code from a UML diagram, the abstraction level is reduced, and the transformation is therefore vertical. On the other hand, exporting an UML diagram to XML keeps the same level of abstraction while changing the modeling language. It is therefore a horizontal transformation.

**Syntactical versus semantical:** Transformations can also differ in whether they just change the syntax of a model or also change the semantics of a model. Syntactical changes are changes like the aforementioned UML to XML export. Semantic changes change the items in the model, for example, when refactoring.

For filling the history graph we generate, we employ iterative graph generators. Each iteration step is a model transformation from the predecessor to the successor. Using the dimensions presented by Mens et al., we can describe those transformations as one-to-one, endogenous, and vertical. The same is true for the merges except that merges are many-to-one.

The goal of modeling frameworks and tools is to automate those transformations. But for automatic transformations, it is important that they meet some standards in order to be useful. These include the well-formedness of the target model, traceability of changes, scalability, and standardizations. This makes it important to rigorously test and benchmark the tools. To develop those tests and benchmarks, synthetic model data is needed.

### 2.1.3 Model evolution

From development to shutdown, software goes through different stages. Bennett et al. [Ben00] introduced a staged model of the software lifecycle shown in Figure 2.2. Evolutionary software

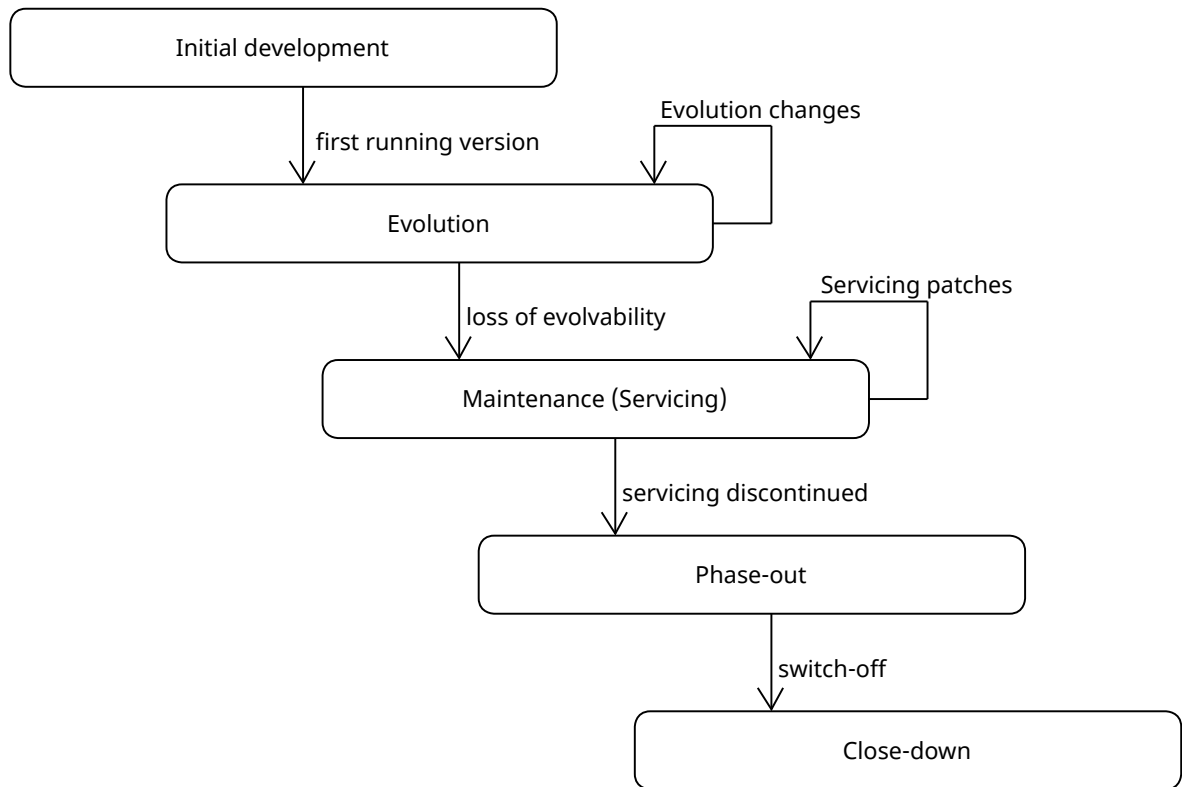


Figure 2.2: First introduced by Bennet et al. [Ben00] this staged model describes the lifecycle of a software. Starting with the initial development, a first running version is created. In the next stage, evolution, developers continuously add new features or react to changes in requirements. Once the software is deemed not evolvable anymore, it enters the maintenance stage. It will get service patches, but the feature set stays the same. When the service patches end, the software enters the phase-out stage. Dependent projects are prompted to switch to a newer version or find other alternatives. Finally, after a grace period, the software is in the close-down stage.

development is a practice where the bulk of the development process is shifted from the initial development towards the evolution stage of a software [Raj14]. Popular examples for evolutionary software development are agile development and open source development. Agile development is defined by short iterative milestones. Between each milestone, stakeholders can assess the software, and their feedback is incorporated into future milestones. A common version of agile development is Scrum [Raj14]. Open-source development distributes the development to multiple volunteer developers. A central developer denies or accepts their contribution. This creates small evolutionary changes that continuously improve the existing software.

In MDSE features can be linked to the metamodel. This means changes in the feature set might require changes in the metamodel. As a result, every model represented in that modeling language will need to adapt to the new version. Van Deursen et al. show four dimensions of evolution in the context of MDSE [vVW07].

**Regular evolution:** The modeling language is used to implement the changes. The modeling language itself and associated tools or frameworks stay the same. This is the case for the changes in the history we generate with our algorithm.

**Meta-model evolution:** The modeling language needs to be adjusted. This may require a migration of the models expressed in that language. This can happen if a project designs its own domain-specific language. If the change only affects a part of the language, it might not

be necessary to migrate all models.

**Platform evolution:** The target platform with associated tools or frameworks change. If the modeling language of the new tools is the same as before, the models do not need to be migrated. Code generators and application frameworks need to change to accommodate the new platform.

**Abstraction evolution:** A new modeling language is added or replaces an existing one. Models will need to be migrated to the new language. This can mean a new domain or technical space is added to the project. While not all models need to migrate, those concerning the new domain need to use the languages of the new domain.

## 2.2 Synthetic Data Generation

Synthetic data generation is the process of algorithmic creation of artificial datasets that replicate the structure and statistical properties of real data [RA23]. This allows the use of data-intensive tasks, like testing, benchmarking, or training models, when data cannot be obtained from real data sources. This can be due to legal issues, privacy concerns, incompleteness, the cost of acquiring, or other similar issues. Due to the variety in different data types and different software ecosystems, there also is a diversity of different approaches for synthetic data generation.

### 2.2.1 Test Data Generation

As a subset of synthetic data generation, there is also test data generation. It introduces a secondary concern to the generated data. The data is not intended to have realistic properties but needs to satisfy specific coverage criteria. The goal is to reach specific parts, like code statements, decision branches, or execution paths [McM04].

Ferguson et al. [FK96] discuss the generation of test data by analyzing the software that is tested. They define their generators as algorithms that find a set of input data that leads to the execution of a previously selected statement in the software. They identify three types of test data generators.

**Random** test data generation just uses random values and checks whether the target statement is executed.

**Goal-oriented** test data generation uses the control flow and execution time values to check whether the target statement can be reached. Whenever an undesired branch is reached, the recorded execution time values are used to refine the new input data.

**Path-oriented** test data generation works similarly to the goal-oriented approach. But before execution, a list of subgoals is generated that needs to be reached before reaching the target statement.

### Fuzzing

A different approach involving test data generation is called fuzz testing. It focuses on finding edge cases using a variety of input data to expose vulnerabilities and bugs. Fuzz testing, also called fuzzing, is not limited to test data generation but also uses test case generation. Figure 2.3 shows the architecture of fuzzers. They can be split into three categories, depending on how much information about the software under test they need [Jää16].

**Black-Box Fuzzing** has no information about the target software. It uses well-formed sample input and a set of predefined mutation rules to generate test data. Given the lack of knowledge about the software, black-box fuzzers often struggle to reach more nested test cases [Lia+18].

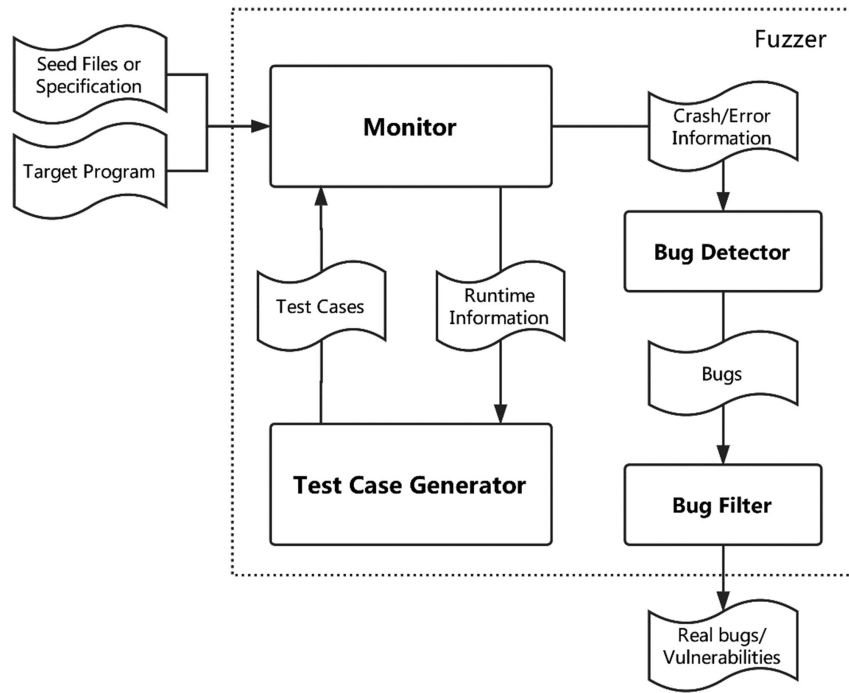


Figure 2.3: Liang et al. [Lia+18] use this diagram to show the general process of fuzzing. The **Target Program** is the software under test. Often the source code is not available. The **Monitor** is the part of the fuzzer that collects runtime data from the software. Black-box fuzzers do not need a monitor. The **Test Case Generator** is responsible for generating the test inputs. Fuzzers usually try to generate semi-valid input data. The data needs to be valid enough to pass internal validation checks but invalid enough to trigger bugs or vulnerabilities deeper in the software. The **Bug Detector** collects and analyzes the resulting crashes to find possible bugs. Since not all bugs are relevant to the tester, there is a **Bug Filter**. This is often done manually.

**Gray-Box Fuzzing** has some runtime information like code coverage and taint data flow. This information is used by algorithms like a genetic algorithm or taint analysis to modify the test data to get more code coverage over the target software [Lia+18].

**White-Box Fuzzing** has access to the source or bytecode of the target software as well as the runtime information. By collecting all conditional statements along the execution path, white-box fuzzers can, in theory, achieve 100% coverage. In practice, they usually fall just slightly short of 100% [Lia+18].

### 2.2.2 Generative Models

Modern approaches often rely on generative models to generate different kinds of synthetic data. The general workflow for generating synthetic data using generative models is shown in Figure 2.4. Rusum et al. [RA23] identify the following three most common approaches for synthetic data generation using generative models.

**Variational Autoencoders** generate new data by training an encoder to map the samples into a latent space and a decoder to reconstruct the sample from the latent space. In contrast to normal autoencoders, the latent space of a variational autoencoder is probabilistic (often a Gaussian distribution).

**Generative Adversarial Networks** are two opposite neural networks. In this case, there is a generator and a discriminator. The generator and discriminator both learn simultaneously, with the generator trying to generate real-looking samples and the discriminator trying to dis-

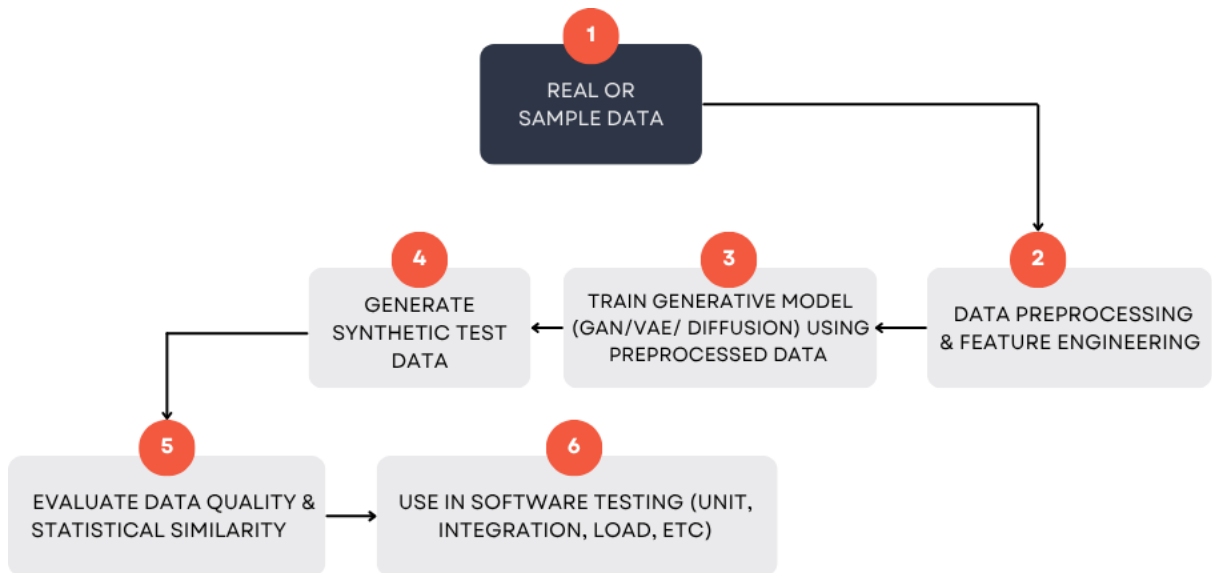


Figure 2.4: Workflow of synthetic data generation with generative models as depicted by Rusum et al. [RA23]. In step one, real data is procured. In the next step, that data is preprocessed and checked for desired features and sample size. Step three is the training of the generative model on the selected data. With the trained model, a set of synthetic data can then be generated. Before newly generated data is used for testing or benchmarking, the statistical similarity to the real data is confirmed.

tinguish between the samples of the generator and the real samples.

**Diffusion Models** use the process of denoising to generate new data from random noise. By progressively adding noise to the real samples, a neural network is trained for denoising. This step is repeated until the model can generate data from pure noise.

## 2.3 Version Control Systems

A version is a specific state of an evolving item [CW98]. The version graph links those versions together. Hereby represents a link from version  $V_1$  to version  $V_2$  that version  $V_2$  was derived from version  $V_1$ .

As seen in Figure 2.5, there are three main structures for version graphs [CW98]:

- Sequence:** A straight progression from one version to another. This is used for very restrictive versioning, usually when each version is a release.
- Tree:** Also called branching, the version graph splits up into two parallel evolving paths. Common pattern when older revisions are still maintained.
- Acyclic graph:** Similar to the tree, but parallel versions are merged together at a later stage. Common for distributed development, where multiple local versions are maintained and regularly merged together.

Version control systems can be either centralized or distributed.

In a **centralized version control system**, one version graph is stored on a central server. Developers connect to that server as a client and need a constant connection to read, write, or edit the version graph [RS16]. A popular centralized version control system is subversion [CM13].

In a **distributed version control system**, each developer has a local version graph. A set of changes is exchanged in a peer-to-peer fashion between local copies to synchronize them. This



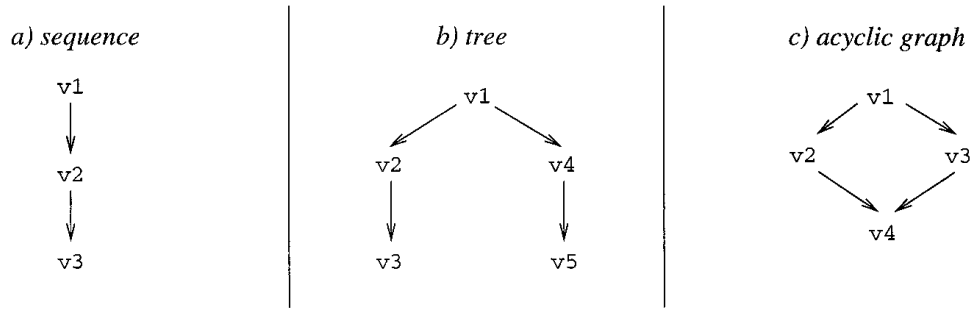


Figure 2.5: Conradi et al. [CW98] show three possible shapes of a version graph. a) sequence: each version has a maximum of one predecessor and one successor. b) tree: each version has a maximum of one predecessor but can have multiple successors. c) acyclic graph: a version can have multiple predecessors or multiple successors. But importantly, no version is able to reach a predecessor by following the directed edges.

means developers only need network access once they synchronize their local copy, allowing for more flexibility than a centralized system [RS16]. A popular distributed version control system is Git [CS14]. In a version control system, each version in the version graph is a commit. A commit is a small set of changes usually accompanied by a log message explaining their purpose or intent [RS16]. A sequence of consecutive commits in a version graph is referred to as a branch. The point at which the graph splits up into two separate branches is called a branching event or a new branch branching off an old one. When two branches combine into a common version, it is called a merging event or one branch merging into another.

Purpose	Description	Num.	%
Bug Fixing	A branch aims to fix bugs.	50	16.7
Feature Implementation	A branch aims to implement features.	125	41.7
Testing	A branch aims to test code, deploy test platform or maintain test cases, etc.	16	5.3
Code Structure Optimization	A branch aims to perform code refactoring or style formatting.	12	4
Documentation	A branch aims to maintain a project's documents, such as its website, license declaration, etc.	10	3.3
Dependency Configuration	A branch aims to declare a project's plugins or dependent libraries' versions.	6	2
Version Iteration	A branch aims to certain version development, prepare for release, and version upgrade. In such a branch, bug fixing and feature implementation are two major activities.	74	24.7
Others	A branch which cannot be put into the above categories goes to this category. E.g., a branch is created for doing nothing.	7	2.3

Figure 2.6: Zou et al. [Zou+19] compiled and categorized the branch use of 2,923 GitHub projects. The three main reasons developers create new branches are feature implementation with 41.7%, version iteration with 24.7%, and bug fixing with 16.7%.

## Branches

Branches can fulfill different roles during development. They can be used to postpone potential conflicts during distributed work on a shared project. This shifts the conflict resolution to the merge stage and allows for smoother development. However, it is more common to have separate branches for each feature of a project. These branches are merged once each feature has been completed.

In a study from 2019, Zou et al. [Zou+19] compiled and categorized the branch use of

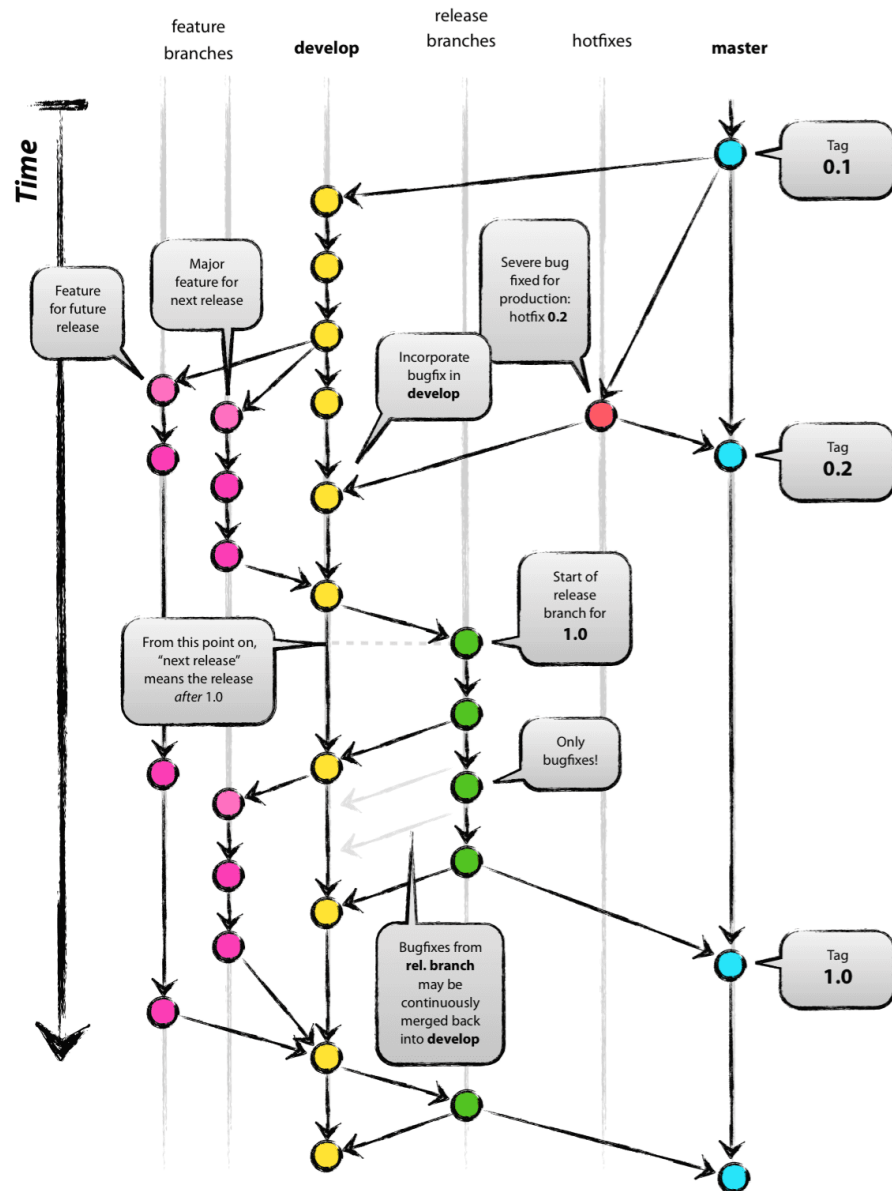


Figure 2.7: Driessen [Dri10] introduced GitFlow. It is one of the most common workflows used with Git. The master branch only contains the fully released versions. The most recent working version is on the develop branch. Every new feature gets its own feature branch. Once a feature is developed, the feature branch is merged into the develop branch. When starting a new release, a release branch is branched off the develop branch. There, bug fixes are pushed until the version is released. Once the release is ready, the release branch is merged into the main branch. The bug fixes of the release branch are also merged back into the develop branch.

2,923 GitHub projects. They concluded that the three main usages for branches are feature implementation at 42%, bug fixing at 17%, and version iteration, which is a mix of feature implementation and bug fixing, at 24% (see Figure 2.6).

This aligns with the findings of Cortés Ríos et al. [CEE22]. They analyzed the usage of different Git workflows and found that GitFlow (see Figure 2.7) is one of the most common workflows. One of the key elements of GitFlow is using feature-specific branches during development. These feature branches are merged into a shared develop branch. Only when a full new release is ready, the changes of the develop branch are merged into the main branch [Dri10].

## 2.4 Graph Generators

There exist a variety of different graph generators. Xiang et al. [Xia+22] propose a classification of graph generators. Figure 2.8 shows a graph representation of the classification. The first differentiation is whether the generator is a traditional graph generator or a deep graph generator. In the following sections 2.4.1 and 2.4.2, we present their findings.

### 2.4.1 Traditional Graph Generators

**Traditional Graph Generators** use graph models to generate a graph based on a small set of parameters. They can be further classified into rule-based generating and block-based generating.

**Rule-based Generators** use a set of explicit operations and parameters to generate a graph. One example is the preferential-attachment graph model, where sequentially new nodes are added. The new node is connected to the rest of the graph with a predefined number of edges.

**Block-based Generators** generate a subset of nodes first. This makes it possible to model more realistic graph degree distribution and clustering than rule-based generators.

### 2.4.2 Deep Graph Generators

**Deep Graph Generators** employ deep learning algorithms to generate graphs. They can be further classified into sequential generating, one-shot generating, and adversarial generating.

**Sequential Generators** rely on an existing incomplete graph to generate new elements. They interpret a graph as a sequence of elements and generate a new element based on the whole

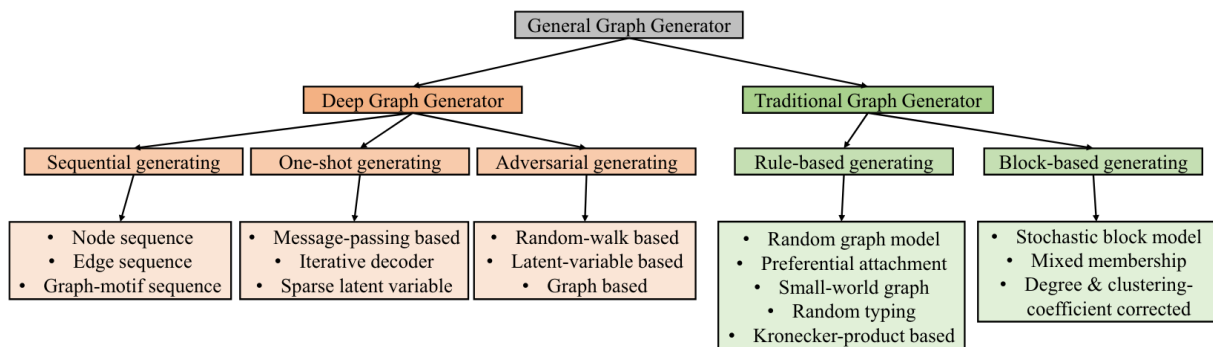


Figure 2.8: Xiang et al. [Xia+22] introduced a classification tree for general graph generators, with their main distinction being traditional graph generators and deep graph generators, where traditional graph generators use rules and graph models to generate graphs and deep graph generators utilize deep learning algorithms to generate graphs.

sequence. This can be a node sequence, edge sequence, or graph-motif sequence. A graph-motif is similar to a block, a group of nodes connected by edges.

**One-shot Generators** generate a whole new graph directly without sequential dependencies. An example of these is an autoencoder-based generators. To generate graphs, they first use a trained neural network to encode a graph into probabilistic parameters. These parameters are then adjusted by adding some random noise. Finally, another trained neural network decodes the parameters back to a graph. The newly generated graph is similar to but different from the starting graph.

**Adversarial Generators** generate graphs by training a generator and a discriminator. This allows for more realistic data but is in general harder to train. They can also be based on autoencoder-based generators. Instead of randomly adjusting an encoded graph, a generator is trained to generate random encodings while the discriminator compares them to encodings of real graphs.

### 2.4.3 Graph modification

Additionally, generators can modify existing graphs. The goal becomes not to generate a new graph but to change an existing graph while keeping certain qualities the same. Those generators can be split into edge transformations and edge-node co-transformations [GZ23]. For edge transformations, the set of nodes in the graph remains the same while new edges between them are generated. Edge-node co-transformation can change the nodes and edges of a graph. An example is an editing-based generator, which generates a sequence of edits on a graph.

### 2.4.4 Open Challenges

In their survey from 2021, Bonifati et al. [Bon+20] identify open problems and challenges in the field of graph generation. Some of those challenges are:

**Simple Usage, Simple Parameters:** Due to the complexity of graph data and the variety of applications for graphs as data, graph generators tend to be highly versatile but also difficult to configure. This can be an issue as users prefer ease of use over configurability.

**Generating Noisy Graphs and Graphs with Anomalies:** To remedy the fact that graph data is generally difficult to de-noise, machine learning algorithms have been adapted to work with noisy graph data. Alternatively, there are machine learning approaches to denoise graphs. To test and benchmark both of those, it is important to generate realistic noisy graph data.

**Evolving Graph Data:** Over time the structure of applications changes due to the environment or use demands. This applies to underlying data and graph data as well. Generating graphs that change over time significantly increases the complexity compared to classical graph problems.

## 3 Related Works

### 3.1 The Need for MDSE Model Data

There are multiple works that highlight the need for software model data.

#### **Machine Learning for Synthetic Data Generation: A Review [Lu+25]**

Lu et al. reviewed the current state of synthetic data generation in the context of machine learning. They focused on different machine learning algorithms used for data generation but also underlined the importance of synthetic data for the training of those machine learning algorithms. While they talk about machine learning in general, this also applies to machine learning in the context of MDSE. They also mention concerns around data scarcity and data privacy of real data. Data scarcity is a very relevant concern for MDSE model data. Data privacy, on the other hand, is less of an issue for MDSE model data, as they usually do not contain private information.

#### **Artificial Intelligence-Driven Test Case Generation in Software Development [Moh25]**

Mohapatra explores the current state of test case generation in the context of artificial intelligence. They explain that detailed software models are an important part of white-box test case generation. Since most traditional software projects do not have high-fidelity models, a generator is used to generate the model automatically from the source code. They also mention the rising influence of machine learning algorithms for data synthesis. Therefore, having an evolving software model with evolving testing needs aids the development of advanced testing toolchains.

#### **Complex Model Transformations by Reinforcement Learning with Uncertain Human Guidance [DD25]**

David et al. presented an approach and technical framework to develop complex model transformations using reinforcement learning supported by uncertain human advice. Being able to use reinforcement learning to autonomously generate complex model transformations can reduce the development time and possible human errors they usually entail. In their discussion David et al. mention the possibility for standalone model transformation chains. Such chains would need a highly varied set of MDSE model transformations. Since procuring such a set from real projects is the only alternative, generating synthetic MDSE model transformations seems to be the more likely approach.

## 3.2 MDSE Model Data Collection

There are approaches to collect existing MDSE model data for the purpose of benchmarking. Some of these models were scraped from sources like GitHub and could therefore have an evolving history. To the best of our knowledge, there is no curated dataset that captures the history of MDSE models.

### **ModelSet: a dataset for machine learning in model-driven engineering [LCC22]**

López et al. introduced ModelSet, a collection of 5,466 Ecore and 5,120 UML models that are labeled with main and secondary interests. Additionally, they introduce a tool to improve the labeling effort. The tool groups models that are very similar, reducing the manual effort when labeling. They stress the need for high-quality data sets for machine learning algorithms, especially labeled datasets. To improve their work they state that they want to move towards crowdsourcing to increase the size of the dataset and the label quality.

### **Toward a Community-Curated Golden Dataset of UML Models [Ver+25]**

Verbruggen et al. present an approach for creating a community-curated dataset of UML models. To create their "golden UML dataset," they first created an initial dataset of UML class diagrams. Then, to grow the dataset, they propose a protocol to publish models to the dataset and review them before they are accepted. They argue that evaluating generative AI for modeling tasks will need a hand-curated dataset, like the one they propose, because generative AI will always be able to produce an output, leaving the question of quality of the output as the deciding factor.

## 3.3 Graph Generation

There are multiple solutions to generate graph data, but to the best of our knowledge none support the generation of evolving graph data.

### **Generating Large EMF Models Efficiently [Nas+20]**

Nassar et al. propose a rule-based approach for generating models that conform to a specific metamodel. Their approach focuses on the Eclipse Modeling Framework (EMF). They broke down the generation task into two subtasks. First the metamodel is used to generate a rule-based model transformation system. Then, as the second task, these rules are consecutively applied to generate the model. They implemented their approach as two Eclipse plugins, with one plugin handling the first task and another handling the second task.

### **Automated generation of consistent, diverse and structurally realistic graph models [Sem+21]**

Semeráth et al. present a technique to create synthetic domain-specific graph models. They collected structural graph metrics of real graph models from three different domains. They interpret graph generation as a state space exploration by using the Viatra solver framework and model extension operations. Using the aforementioned metrics, they guide the exploration, using a hill-climbing strategy, towards their target metric. Figure 3.1 shows the process they use to generate these models.

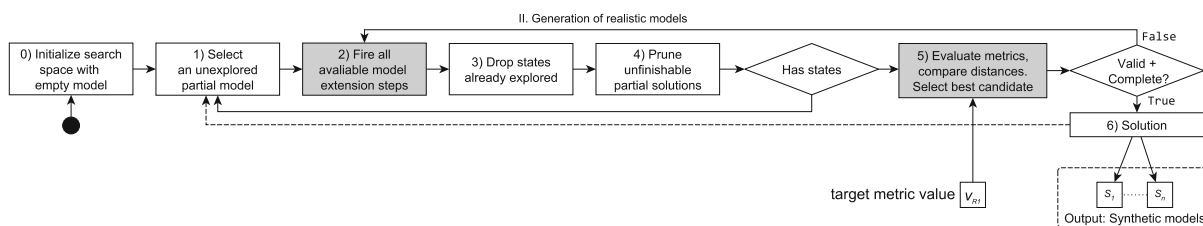


Figure 3.1: The generation process of model graphs by Semeráth et al. [Sem+21]. They extend an initially empty model using their model extension steps. By repeatedly applying all possible extensions and selecting the candidate with the most promising metrics, they are able to generate models with realistic features.

### Accurate and Consistent Graph Model Generation from Text with Large Language Models [Che+25]

Chen et al. proposed a framework to enhance the already existing capabilities of LLMs to generate MDSE models from natural language. To achieve this, they aggregate multiple outputs from the same LLM. They show that this increases the overall quality and consistency of the generated models. While not directly mentioned by Chen et al., this could be a useful tool for test data generation as well.

### Generative Code Modeling With Graphs [Bro+19]

Brockschmidt et al. present a model for source code generation by using graphs to represent intermediate states of the generated output. They use the syntax tree and enrich it with additional edges to describe the flow of information. This turns the problem of code generation into a problem of graph generation. Their approach involves "hole completion," where a trained machine learning algorithm tries to fill holes in an otherwise completed program using the graph representation to ensure syntactic and semantic cohesion.

### Optimization of Molecules via Deep Reinforcement Learning [Zho+18]

Zhou et al. present a framework for molecule optimization using reinforcement learning. They represent molecules as graphs with nodes being atoms and edges being bonds. To modify the molecules, they use a Markov decision process. This breaks the modification down into three types of modifications: atom addition, bond addition, and bond removal. This achieves the certainty of chemical validity. We approach the graph generation in a very similar way, by defining fundamental actions on the graph and applying them sequentially.

## 4 Conceptualization

This work builds on the GraphGentool introduced by Kegel et al. [Keg+24]. It utilizes the GraphGentools graphcore module for the graph representations of MDSE models. As seen in Figure 4.1, a graph consists of an id, a collection of nodes, and a collection of edges. A node has an ID, a name, and a collection of attributes. Each node is either a simple node with a color or a region node with a subgraph. Graph  $G_1$  depends on  $G_2$  when  $G_1$  contains a region with  $G_2$

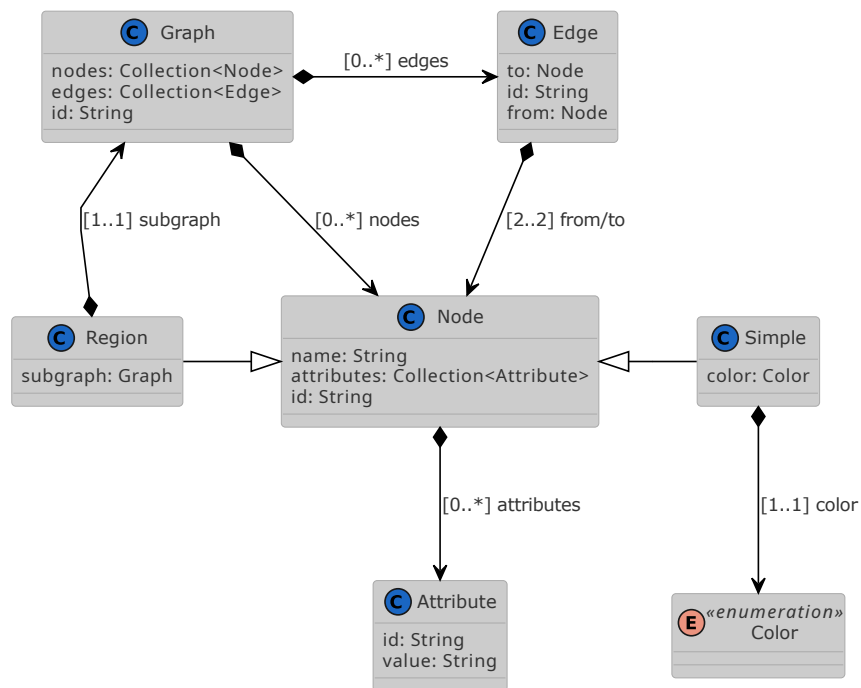


Figure 4.1: The UML class diagram for the model graphs. Each graph has a collection of edges and nodes. Edges are directed and point to their source and target nodes with the from and to properties. The node class is abstract. There are two implementations of the node class. Simple nodes have a color. Region nodes have a subgraph. The subgraph of a region cannot contain the region itself or other regions whose subgraphs contain the region. In other words, the dependency between regions can never be circular. Additionally, all nodes have a list of attributes. Graphs, edges, nodes, and attributes all have an ID. This is important to identify equal elements when importing or exporting to other formats, and it reduces the complexity of finding the difference in two coevolving graphs.



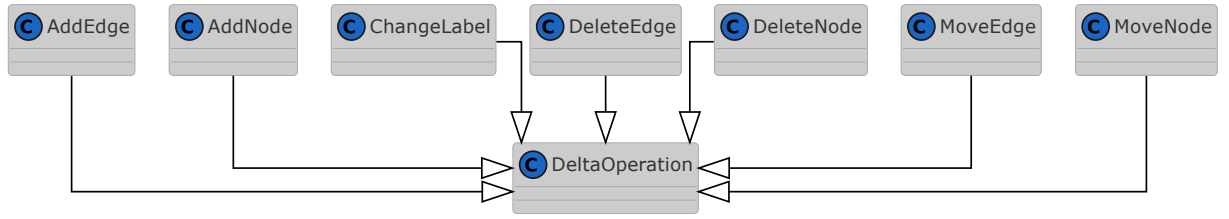


Figure 4.2: The UML class diagram for the graph deltas from the deltacore module. There are seven different atomic operations on the graph model. Those operations are adding an edge, adding a node, changing the color of a node, deleting an edge, deleting a node, moving an edge to a different region, and moving a node to a different region. Since the descriptor of the simple node recently changed from label to color, the atomic action is still called *ChangeLabel*. This will probably change in future versions. In addition, the deltacore module also contains a *DeltaSequence*, which is used as an ordered collection of delta operations.

as the subgraph or if  $G_1$  contains a region with a subgraph that depends on  $G_2$ . Cyclic dependencies are not allowed. This allows recursive collection of nodes and edges throughout all subgraphs. An edge is directed and has a from node and a to node and also an ID. Edges can span from one region to another.

Additionally, we use the deltacore module. This module provides a set of atomic operations performed on graphs of the graphcore module. Figure 4.2 shows the seven kinds of operations. *AddEdge* describes the addition of an edge between two existing nodes. *AddNode* describes the addition of a new node. If the new node is a region, the subgraph is empty. *ChangeLabel* describes the change of the color of a simple node. *DeleteEdge* describes the deletion of an edge. *DeleteNode* describes the deletion of a node. Also contains a list of implicated node and edge deletions if the deleted node was a region with a non-empty subgraph. *MoveEdge* describes the move of an edge from one region to another. *MoveNode* describes the move of a node from one region to another. To collect these operations, the deltacore module provides the *DeltaSequence* class. This is an ordered collection of *DeltaOperations*.

## 4.1 Requirements

As stated in Section 1.3, we want to design our generated history so it is independent of the model generator. This will also help us to achieve RQ2, which is important to achieve a high degree of customizability in what kind of models can be generated. To generate realistic histories (RQ1), we want to differentiate between different branch types based on their purpose. To keep the generation process simple, we limited it to three types of branches: the main branch, refactor branches, and feature branches. Additionally, the process of history generation needs to be parameterized. The first parameter is a random seed that is used for all pseudo-random actions during the generation. This is important to be able to replay the generation and should be automatically passed down to all instances where a random function is used. Next is the number of commits. This defines the size of the history. There also is the commit size parameter to define how many atomic actions are performed in each commit. The next five parameters define the shape of the history. Those are the branch length, the maximum amount of parallel branches, the minimum amount of parallel branches, the median amount of parallel branches, and a spread value defining how clustered the branch amount is around the median. Another parameter is the chance that new branches spawn from branches other than the main branch. There also is a ratio for the branch type of the new branch. Next we have the chance that a branch is never merged back into the main branch. Since workflows like GitFlow [Dri10] usually discourage commits on the main branch, we also have parameters for

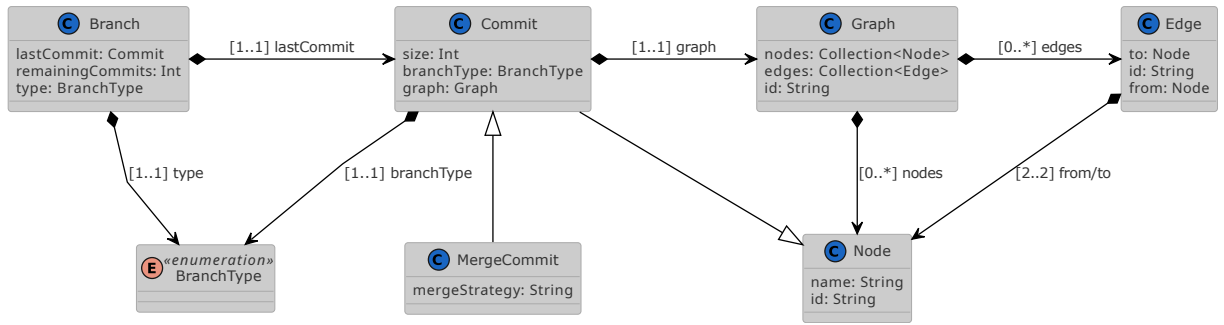


Figure 4.3: The UML class diagram for the history graph. The graph, node, and edge classes are the same as in the model graph from Figure 4.1. Instead of simple and region nodes, we have the commit that extends the node. Each commit has an integer size that describes the amount of atomic actions since the last commit, a branch type of the branch the commit belongs to, and the model graph at the current point of the history. Branches are objects that only exist during the history generation. They store how many commits they are going to contain and point to their latest commit, allowing the generator to gradually fill them by adding new commits to one branch at a time. When a branch ends, it can be merged back into the main branch. To accommodate that, there is the merge commit. Merge commits are the only commits with two predecessors. They contain a merge strategy string that specifies the merger used for generating the merged model.

the main branch and subbranch priority, which determine how likely the respective branch will get assigned commits. To generate all the IDs for the history graph, there is a name sequence as a parameter. Then we have a list of preconfigured generator adapters that allow the history to employ different generators for populating the history with model graphs. Those are a main branch generator, a feature branch generator, a refactor branch generator, and finally, a merger that can generate a merged model graph. Each version step is generated by those generators using the previous model graph, so we also want an initial model graph as a parameter. Since each generator also needs to be configured, the total amount of configuration puts a considerable burden on the user. To make it more usable, we define some useful default values as well as provide a factory method that adjusts the other values based on a few provided ones. We go more into detail in section 5.6.

## 4.2 History

We want to generate a generic history that could be produced by different version control tools. This means we need to produce an acyclic graph of versions with directed edges between them showing their progression. To do this we extend the classes from the graphcore module, as seen in Figure 4.3. Following systems like Git, we make each version of the version graph a commit. Each commit contains the model as a graph and an integer size, describing how many delta operations are needed to transform the model graph from the previous commit to the model graph of this commit. Merge commits are the only commits with two predecessors. Therefore, there can't be one integer number of delta operations from the previous commit. Instead, merge commits describe a merge strategy used to reach a unified model based on the models of their two previous commits. To ensure that the history graph is acyclic, we use branches. Each branch points at its most recent commit, and new commits can only be added at the end of a branch. Additionally, it holds a value of commits that still need to be generated before the branch is merged. As shown by Zou et al., branches usually have a purpose [Zou+19]. To reflect that, we introduce three branch types: main branch,

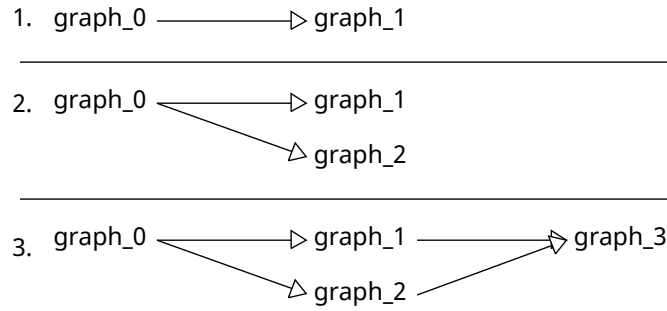


Figure 4.4: Example steps for generating a branching and merging event. It also shows the three possible actions during the history graph generation. In step 1, a model graph generator is used to generate `graph_1` using `graph_0`. This is a continuation, because `graph_0` did not already have a successor during the generation. In step 2, `graph_0` is used to generate `graph_2`. This is a branching event, because `graph_0` already had `graph_1` as a successor. In step 3, `graph_1` and `graph_2` are merged together using the merger. This is a merge event. Merges are only allowed if both graphs do not have a successor yet. Using these actions, we can create diverse histories. During the generation, the branch object defines what action is taken next to ensure that the history conforms with the configured parameters.

feature branch, and refactor branch. Commits in feature branches should contain primarily commits with positive deltas (more `AddNode/AddEdge` than `DeleteNode/DeleteEdge`), while commits in refactor branches should contain mostly neutral deltas (equal `AddNode/AddEdge` and `DeleteNode/DeleteEdge`). We achieve this by having different generators for feature and refactor branches in the configuration. While there can be any amount of feature and refactor branches, there is only one main branch. This is where the progress made in the other branches is collected. Once a branch that is not the main branch has depleted its expected amount of commits, it is either merged into the main branch or declared abandoned. By configuring an expected range of parallel branches, a range of expected branch lengths, and an amount of total commits, we can define the overall shape and interconnectedness of the history graph.

### 4.3 Adaptability

In this section we will explore [RQ2](#) as we discuss how we will integrate existing model graph generators into our algorithm.

Once the history is generated, we need to populate it with model graphs. Since we want to be able to use different kinds of generators for the model graphs, we need to define an interface that allows us to interchange them. Given the structure of the history, we need each generator to be able to generate a set of changes on top of an existing graph. This is a typical task for sequential graph generators. In addition to the number of atomic actions and the previous model graph, each generator has its own configurations. Therefore, the generator is wrapped in an adapter object that contains all necessary configurations for the generator and is reusable, so it can be handed down throughout the generation process.

Each generator is left with the task of generating a list of delta operations that transform an input model graph  $g_i$  into an output model graph  $g_o$ . Using the  $g_o$  graph of the previous commit as the  $g_i$  of the next commit, these generations can be chained together to form a sequential history. When there is a branching event, two commits share one predecessor. In that case both of them use the same model graph as their  $g_i$ . This means the generator needs to be

pseudo-random and not be purely deterministic on  $g_i$  so that the two commits can produce different  $g_o$  graphs. When two branches merge together, we now have two different  $g_i$  that need to be combined into one  $g_o$ . Since graph generators are not designed to generate a graph from two input graphs, we need to define a new adapter that is responsible for merging. Similar to the generator adapter, the merger adapter also needs to be reusable and can be configured based on the needs of the specific merger in question. This allows us to use existing tools for model merging during the generation process.

To initialize the generation there needs to be an initial commit that has no predecessors. This means we can not use the generator to generate its model graph. Instead, we use an initial model graph  $g_0$  that is provided to the history via the configuration. We can still generate the entire model graph by making  $g_0$  the empty graph. Alternatively, we can use another graph generator to produce the initial model graph. This makes it possible to incorporate one-shot generators, or other non-sequential graph generators, and build a history that continues on top of their model graph.

To handle the difference between the branch types, we require three different generator objects. This means the configuration for the history generation will contain a high amount of parameters. As Bonifati et al. [Bon+20] pointed out, ease of use is an important concern for test data generation. This means we should create an option to fully configure the history generation with just a few parameters. We can achieve that by defining meaningful default values and defining factory methods that take care of parameters that are dependent on each other.

# 5 Implementation

## 5.1 Abstract Algorithm

We use the HistoryBuilder class to create an easy all-in-one object to generate the model graph history. The three methods this class provides are the following:

**generateHistory:** generates the history with empty commits, meaning the model graph of every commit is equal to the empty graph.

**populateHistory:** fills the empty history with actual model graphs.

**exportHistory:** allows to write the history to a file.

A configuration needs to be provided when creating the builder object. This configuration is then passed down to all other objects used to create the history. Figure 5.1 shows the general workflow of the history graph generation and population. The first task of generating the empty

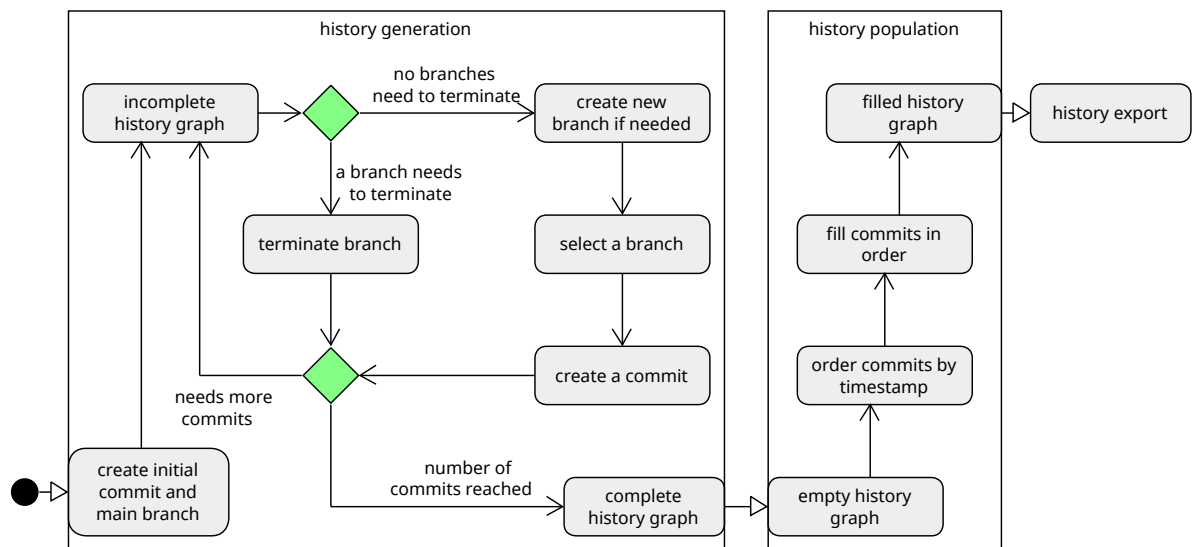


Figure 5.1: General generation workflow of our history graph generator. The workflow is split into two main parts: the history generation and the history population. During the history generation, initially, the main branch and the initial commit are created. Then, iteratively, new branches are added and filled with commits until the desired number of commits is reached. During the history population, the commits are ordered by timestamp, and then model graphs are generated for each commit in that order. Finally, the fully populated history can be exported.

```

1  while(remaining_commits > 0):
2      if branches.check_finished():
3          selected_branch = branches.get_finished()
4          if selected.needs_merge():
5              new MergeCommit(selected_branch, main_branch)
6              remaining_commits -= 1
7              branches.delete(selected_branch)
8              continue
9      if branches.check_new_brach():
10         branches.add(new Branch())
11     selected_branch = branches.get_random_from_priority()
12     new Commit(selected_branch)
13     remaining_commits -= 1

```

Figure 5.2: Pseudocode of the main loop in the history generation. A variable tracks the commits that need to be generated. Once the desired number is reached, the loop finishes. During each iteration, it first checks to end empty branches, then creates a new branch if needed, and finally adds a new commit to the history.

history is handed down to the HistoryGenerator class. This class contains the main loop of the history generation. Before starting the loop, the generator first creates an initial commit and the main branch containing the initial commit. Also, it creates a list of currently active branches before starting the main loop.

### 5.1.1 History Generation

Figure 5.2 shows the structure of the loop. This loop runs until the desired amount of commits, specified in the configuration, has been created. First it checks if any branch has run out of open commits, meaning there are no more commits needed to be generated for it to reach the desired amount of commits in the branch. If that is the case, depending on whether the branch will merge or not, a merge commit is created on the main branch. Then the branch is deleted and the loop jumps to the next iteration. If no branch is terminating, the next check is whether a new branch needs to be created. No matter if a new branch was created or not, one branch is chosen that will get a new commit in this loop iteration. This is safe to do, as new branches need to have at least one open commit, and any old branch that has no open commits will be terminated and removed from the list of active branches in the beginning of each loop iteration. Finally, the new commit with an empty graph is generated, and the loop can go to the next iteration. While generating those commits, each of them gets a relative timestamp, denoting at which iteration they have been generated. This gives all commits an order, where the timestamp of a commit  $C_x$  is always smaller than another commit  $C_y$  if  $C_y$  can be reached from  $C_x$  when following the edges of the history graph.

### 5.1.2 History Population

The population of the history is handed over to the HistoryPopulator class. First the model graph of the initial commit is set to the initial model graph provided by the configuration. Then the order of the commits is used to iterate over them all. The generator, provided in the configuration, can then generate the model graph of the current commit by using the model graph of the commit that comes before in the history and the size of the current commit. Since the population starts with the lowest timestamp and goes up after that, for each commit, the previous one has already been filled with a generated model graph or is the initial commit

that has been filled before the iteration started. If the commit is a merge commit, instead, the merger provided in the configuration is used.

### 5.1.3 History Export

The export is a simple mapping from the history graph Kotlin classes to JSON. As the generated JSON will contain the whole model graph for each commit, this will be memory intensive when done conventionally. To prevent memory overflows, we therefore need to use stream-based marshalling.

## 5.2 Branches

Branches themselves are objects that only exist during the history generation. They are not saved into the history graph and are just an emerging property, where a sequential section of the history graphs is an individual branch.

### 5.2.1 Spawning New Branches

The configuration provides five values determining how often a new branch needs to be spawned. Those are: A branch length, which uses the random classes described in Section 5.3 to give a specified integer length for new branches. An integer for the maximum amount of parallel branches, another one for the minimum amount of parallel branches, and a last one for the median amount of parallel branches. Lastly, there is a double value between 0 and 1 determining the spread from the median. When checking whether a new branch should be spawned, first we check that the amount of active branches is currently between the minimum and maximum values. If the amount of active branches is smaller than the minimum, we always want to spawn a new branch. If the amount of active branches is bigger than or equal to the maximum, we never want to spawn a new branch. Otherwise, we use the mean of the branch length to calculate the base spawn rate, at which we need to spawn new branches to sustain the amount of active branches. If the spread value is equal to 1, we use that rate to generate a random boolean determining whether a new branch is spawned or not. Otherwise, we use the spread value to adjust the spawn rate before generating the random boolean. If the current amount of branches is higher than the median amount of branches, we use the following formula to adjust the base spawn rate:

$$newRate = baseRate \cdot \left( \frac{(currentB - medianB) \cdot spread + maxB - currentB}{maxB - medianB} \right)$$

If the current amount of branches is lower than the median, we use this formula instead:

$$newRate = baseRate \cdot \left( \frac{(medianB - currentB) \cdot (2 - spread) + currentB - minB}{medianB - minB} \right)$$

This has the effect that the further the current amount of branches is over the median amount of parallel branches, the lower the spawn rate gets, and the further the current amount of branches is below the median amount of parallel branches, the higher the spawn rate gets. The speed at which the spawn rate grows is determined by the spread value. The maximum possible spawn rate is double the base spawn rate if the spread value is 0 and the current amount of branches is equal to the minimum amount of branches. The minimum possible spawn rate is 0 if the spread value is 0 and the current amount of branches is equal to the maximum amount of branches. This adjusted spawn rate is then used to generate a random boolean that determines whether a new branch is spawned or not.

### 5.2.2 Branch variables

When spawning a new branch, all of its values, which determine its behavior, are already set. Those are:

The **latest commit**: it is important, as during the generation of each commit we also need to create an edge from the previous commit to the new one. When creating a new branch, this value is set to the last commit of the branch it is starting from. This variable will change as new commits are added to the branch.

The **branch type**: this can either be a main, feature, or refactor branch. This value is passed down to the commits generated for this branch and is later used to determine which generator

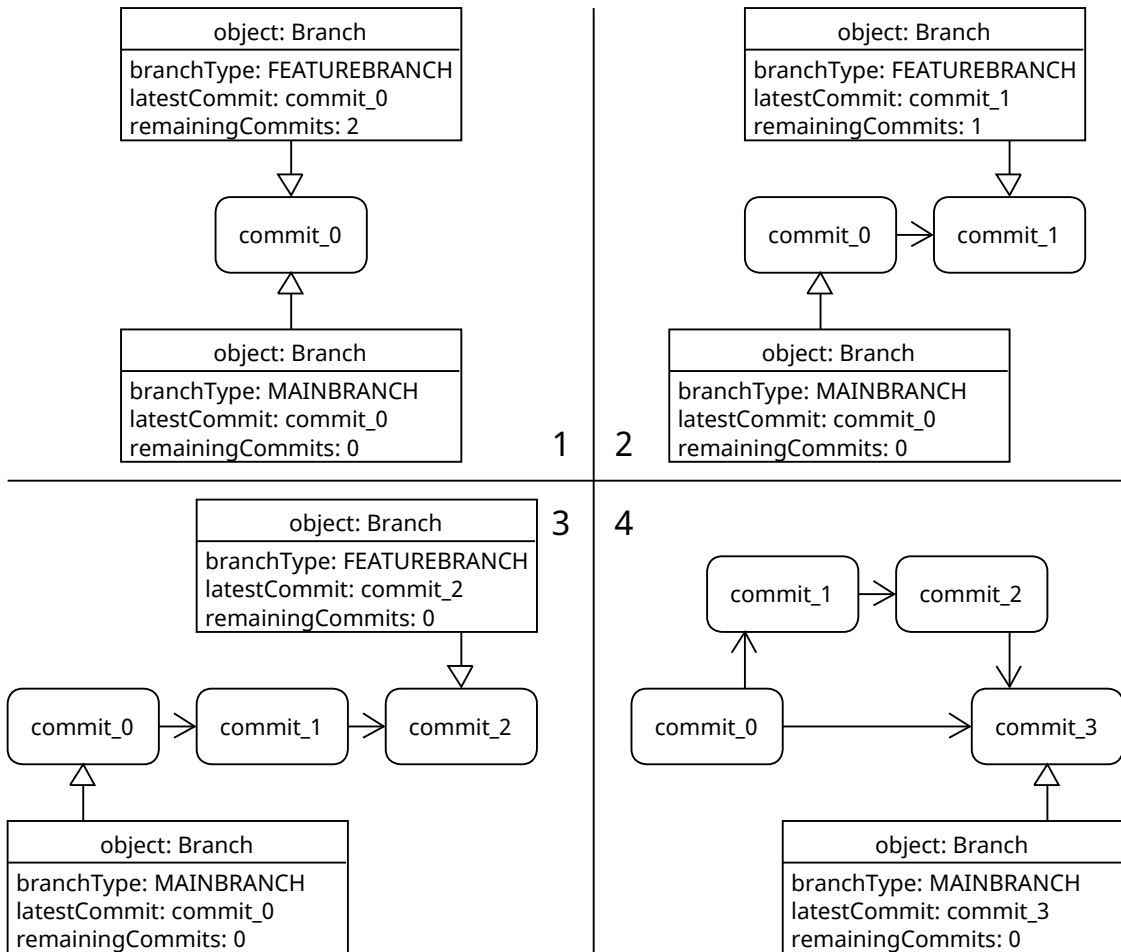


Figure 5.3: Example showing the role of the branch objects during the history generation. At step 1, the generator creates a new branch. During the creation of the branch object, values like the branch type, but also whether the branch merges or not and how many commits, will be in the branch, are set. Both the main branch and the new feature branch point at the same commit as their latest commit. In step 2 and 3, new commits are generated for the feature branch. The latest commit of the main branch stays at commit\_0 while the feature branch moves on. With each new commit the value of remaining commits in the feature branch is reduced. In step 3, this value reaches 0. This means in the next step the branch will get terminated. In this case it is done by creating a merge commit, merging the branch back into the main branch. Once the branch is finished the object gets deleted.



should be used for generating the model graph of the commit.

The **branch priority**: this value is used to determine how likely it is that a commit for this branch is generated. Feature and refactor branches share the same priority.

The **remaining commits**: this variable is reduced by one whenever a new commit is added to the branch. Once this value reaches 0, the branch is considered done, and no new commits will be generated for it. This value is ignored for the main branch.

The **branching boolean**: this determines if a new branch will start from this branch. Once a branching event happens on this branch, the value is set to false unless it is the main branch, which can have an infinite amount of branching events.

The **dead boolean**: this determines whether the branch is merged into the main branch once it runs out of remaining commits.

Figure 5.3 shows how the latest commit and the remaining commits variables change during the lifetime of a branch.

## 5.3 Random

Since we need all of our random values to be seeded for replayability, we extended the Kotlin Random class to the SeededRandom class. This allows more clarity that a specific seed is needed for all random actions and adds the utility to retrieve the random seed at a later point. We also want to make sure that we use the same random generator throughout the generation process. This is because two random generators with the same seed would generate the same sequences of values and could therefore introduce unwanted dependencies between random values. With this in mind, we designed a RandomSequence class that allows us to configure the history generator with different kinds of random distributions using one shared base random generator.

### 5.3.1 The Random Sequence

The base structure of every RandomSequence is that it has an internal random generator that can be replaced, a next method that provides the next value in the random sequence, and a getMean method that provides a double value that represents the mean of all values in the sequence. The RandomSequence has a generic type, allowing one to specify what type of value should be provided. This way we can specify that, for example, the branch length needs to be a random sequence of integers.

We also implemented a few examples of those random sequences, but they can be easily extended if needed. The classes we implemented are:

The **RandomBooleanSequence**: this gives random true or false values and takes a boolean as the rate of how often true should be returned. The mean is equal to the true rate and can be considered the mean if true is 1 and false is 0.

The **RandomUUIDStringSequence**: this provides random UUID strings. Since there is no meaningful mean for strings, the getMean method always returns 0.

The **RandomUniformIntegerSequence**: this provides integers between a given min and max value. The distribution is a normal distribution, and the mean is the average of the min and the max value. This is basically just a wrapper for the standard nextInt method of the Kotlin Random class.

The **RandomSimpleCLTNormalDistributionSequence**: this provides normally distributed doubles with a mean of 0 and a standard deviation of 1. This is achieved using the central limit theorem to approximate a normal distribution with an Irwin-Hall distribution with a number of predefined samples. The default value for the sample number is 12, which gives a fairly accurate approximation and limits the outliers to  $\pm 6$ .

The **RandomCLTNormalDistributionIntegerSequence**: this provides normally distributed integers with a provided mean and standard deviation. To do this, it uses a **RandomSimpleCLTNormalDistributionSequence** and shifts, scales, and rounds the double values to match the desired mean and standard deviation.

When configuring the history generator, the base random generator of each sequence is set automatically to the shared random generator. This allows for easy configuration while at the same time opens the possibility to introduce more complex random distributions if the need arises. The downside is that with the current implementation the random values are independent of outside factors. This means random values like the spawn rate of branches described in section 5.2 cannot be represented by a **RandomSequence**. If the random sequence is extended to also encompass those conditional random values, the branch spawning could also be represented by a random sequence and allow for more flexibility in the configuration of the history generator.

## 5.4 Graph Generation

The model graph generators can have vastly different parameters, so we wrap them in an adapter class and an adapter configuration class for the parameters (RQ2). This allows us to specify the abstract base adapter class as the type for the generators in the configuration of the history generator. New adapters can then be added by implementing a simple adapter that extends the base adapter. Each adapter only takes its corresponding configuration as a constructor parameter. Once configured, the random generator of each generator is set to the shared random generator of the history. This means if the generator adapter needs to create another configuration for the actual generator, it should not be done during the initialization of the adapter, as the random number generator in the configuration changes after initialization. Each generator adapter has the *generate* method. This method takes a graph and an integer amount of steps as arguments and returns the generated graph and a delta sequence to reach the returned graph from the input graph.

We implemented two example adapters: one for the **GraphGentool** and one for a simple generator we implemented called circle generator.

### 5.4.1 GraphGentool

The original **GraphGentool** already has the capabilities to iteratively change a model graph in the form of variants. We can configure the variant generator of the **GraphGentool** to generate one variant with a distance to the base graph equal to the desired commit size. With this, the variant generator conforms to our design of the adapter. All that is needed is to collect all parameters that are relevant for the variant generation in adapter configuration. These parameters are:

The **atomicCounting** parameter describes whether implicated changes, like the deletion of edges when one of their nodes is deleted, are counted as one action or as multiple. If it is set to true, the variant generator will elect not to perform changes if they would implicate more changes than should be generated. This will skew the edit probabilities towards changes with fewer implications. If it is set to false, the number of atomic edits will be higher than configured in the history generator. The default value is set to true.

The **branchEditFocus** parameter describes the chance that consecutive edits have to take place in the same region. 0.0 means every edit happens in a randomly selected region. 1.0 means that each edit needs to be in the same region as the previous one. Between commits, the last edited region is lost because each commit requires an independent execution of the variant generator. The default value is 0.0.

The **editProbabilities** parameter is a string that stores the probabilities for each possible edit to occur. They are in the order of add a simple node, add a region, delete a node, move a node, change a label, add an edge, and delete an edge. The default value is 15:5:5:5:25:25:20.

These default values are taken from the configuration of the GraphGentool. Parameters that are not configureable but are still used by the variant generator, are the branch length, which is set to the desired commit size, and the random seed. Since we can not pass a random generator to the variant generator, we use the random generator of the history generator to pass a random seed instead. We do this because each execution of the variant generator needs to have independent random values, so we can't just pass the seed of the history generator.

### 5.4.2 Circle Generator

To demonstrate the interchangeability of the graph generator, we propose a second generator we call circle generator. This is a simple motif-based graph generator. The basic principle behind this generator is that we generate a circle of nodes connected through edges. Each node has two edges, one that is starting from the node and one that ends at the node. Then a number of connecting edges is created from a node in the circle to a node in the previous graph. Both nodes need to have only two edges beforehand. Figure 5.4 shows how the circle graph generator adds more circles to the graph. Since we want to make the number of nodes in each circle independent from the number of atomic actions per generation, instead of generating the motif directly, we generate a list of deltas to reach the motif. We extend the graph by giving it two more values, a list of upcoming changes that holds delta operations, and a list of circles that holds lists of node IDs. When generating a new motif, we write the changes into the list of upcoming changes and add the new circle to the list of circles. To run the circle generation, we pull deltas from the list and apply them to the graph. Once the list is empty, we generate a new circle or delete an existing one, which then fills the the list of upcoming changes again. To delete a circle, we simply generate all implicated deletes when deleting the nodes in the circle and add them to the upcoming changes. This makes the circles independent from the amount of generation steps but also leads to incomplete circles at the commit. The history population hands those graphs down from generation step to generation step, so later commits can finish a circle that was started in an earlier step. Only when a mixed generator strategy or a merger that does not keep the circles and upcoming changes in the graph object

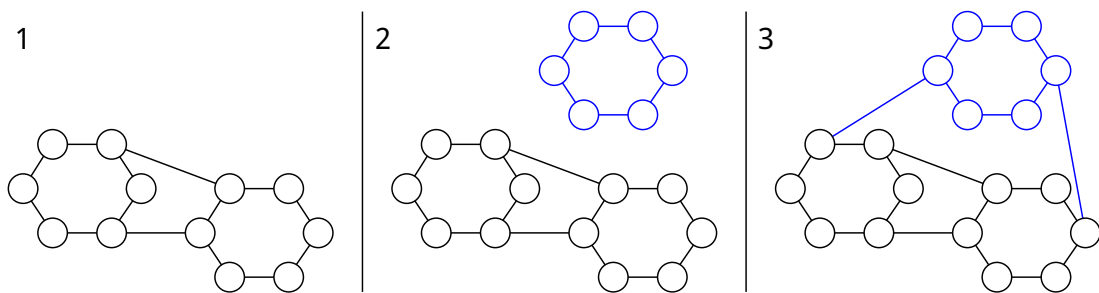


Figure 5.4: Example of how the circle graph generator creates a new motif to expand the model graph. First the generator creates a circle of connected nodes. These nodes can be either simple or region nodes. Then the generator chooses different nodes from the circle and connects them to nodes of the input graph that do not have a connection outside of their circle. Instead of adding those nodes directly to the graph, all new nodes and edges (blue) are instead represented by the delta operations that create them and stored in an ordered list to be applied later.

is used, these partial circles persist and old circles cannot be removed later. The parameters to configure the generator are:

The **numberConnection** parameter is the amount of connection edges each circle gets when it is created. The value is a random integer sequence, and the default value is a uniform distribution between 1 and 3.

The **connectionDirection** parameter describes whether a new connection comes from the new circles or goes to the new circle. This is a random boolean sequence. If this value is true, the connection goes from the circle to the rest of the graph. Otherwise, the connection comes from the rest of the graph to the circle. The default value is a sequence with equal true and false values.

The **circleSize** parameter determines the size of each circle. This value is a random integer sequence with a default value of a normal distribution with a mean of 15 and a standard deviation of 5.

The **nameGenerator** parameter generates the string names of all elements. It is a random string sequence and has a default value of a random UUID string sequence.

The **nestedCircleChance** parameter determines whether a new circle is created in the graph of a region. This is a random boolean sequence. When the value is true, the new circle is in the region. Otherwise, the circle is created in the root graph. If there are no regions, the circle will be spawned in the root graph even if the value is true. The default value is a sequence with a 10% chance of being true.

The **removalChance** parameter decides whether a new circle is created or an old one is deleted. This is a random boolean sequence as well. If the value is true, a circle is deleted. Otherwise, a new circle is created. If the list of existing circles is empty, the generator will always generate a new circle, because only known circles can be deleted. The default value is also a sequence with a 10% chance of being true.

The **regionSpawnChance** determines whether a new node is a simple node or a region. This is another random boolean sequence. When the value is true, the new node is a region. Otherwise, the new node is a simple node. The default value is a sequence with a 2% true chance, meaning 1 in 50 nodes are regions.

We chose to use the random sequences from the history generator, as they allow further configuration and let us reuse the probabilistic logics like uniform and normal distribution. This simplifies the code of the actual generator.

## 5.5 Merge Generation

Similar to the generator adapter, the merger adapter also functions as a compatibility layer. Each merger needs to provide a merge function that takes two model graphs and returns one. Additionally, the merger has the name parameter. This should be a unique name that identifies the algorithm used to generate the merge. This name is used to later identify which merge strategy was used in the history.

For our evaluation we needed a sample merger that is lightweight with minimal impact on the generators. Therefore, we implemented a `RandomRejectMerge`. That merger merges two graphs *A* and *B* by randomly rejecting all changes from either *A* or *B* and completely accepting the other. This is done by simply returning a copy of either *A* or *B* as the result of the merge. The advantage is that this is very fast and even keeps the hidden values of the circle graph generator intact, but it also is not very realistic and therefore not applicable if the goal is to use the generated models as realistic test data.

## 5.6 Default Values

To allow for easy configuration of the generator, we provide default values for all parameters (see Figure 5.6). This makes all parameters optional, allowing the user to only specify values they are interested in. But some of the values depend on each other. For example, the maximum number of parallel branches cannot be smaller than the minimum number of parallel branches. So for users who want to specify only a few parameters, we designed a second factory method called **createSimpleConfiguration**. This method offers less configurability but reduces the amount of parameters to seven. All of these parameters have default values, which again makes them optional. The **seed**, **amountCommits**, **initialGraph**, **mainGraphGenerator**, and **graphMerger** are the same as in the full configuration. The **medianParallelBranches** parameter has a default value that depends on the value passed as the amount of commits. It is the amount of commits divided by 100. The **branchLengthSequence** parameter also has a default value that is calculated using the amount of commits. It is a normally distributed integer sequence, with a mean of the amount of commits divided by 100 and standard deviation equal to the amount of commits divided by 600. The **commitSizeSequence**, **nameGeneratorSequence**, **parallelBranchesSpread**, **subbranchSequence**, **refactorBranchSequence**, **deadBranchSequence**, **mainBranchPriority**, and **nonMainBranchPriority** use the default value of the full configuration but cannot be passed as parameters to prevent cluttering the method. The **featureGraphGenerator** and **refactorGraphGenerator** are set to the same generator as the main generator. The **maxParallelBranches** are set to be 40% above the median, and the **minParallelBranches** are set to be 40% below the median.

Parameter	Value	Description
seed	current time	This seed is used to create the Random instance used throughout the generation.
amountCommits	1000	Fairly low number to give fast generation times even with slower graph generators.
branchLengthSequence	Uniform between 15 and 20	A simple Random sequence with relatively low average have highly interconnected history
commitSizeSequence	Uniform between 2 and 30	High variety to allow for very different sized commits.
nameGeneratorSequence	UUID Strings	UUIDs generated using the base Random instance.
mainGraphGenerator	LegacyGeneratorAdapter with default configuration	The generator adapter for the GraphGentool. It uses the default values of the adapter.
featureGraphGenerator		
refactorGraphGenerator		
graphMerger	RandomRejectMerger	Randomly chooses one of the two input graphs.
maxParallelBranches	15	With the median of 10 and a max and min 5 above and below, there is a healthy spread of different possible amount of branches.
minParallelBranches	5	
medianParallelBranches	10	
parallelBranchesSpread	0.7	0.7 is a good value to mostly stay below the min and max values but still have a variety in values
subbranchSequence	RandomBooleanSequence 10% True	10% of branches will have a subbranch.
refactorBranchSequence	RandomBooleanSequence 30% Refactors	Not relevant if just defaults are used but set to 30% so there is an effect if a different generator for refactor and feature branches is set.
deadBranchSequence	RandomBooleanSequence 10% True	Most branches still merge but there are some that do not.
mainBranchPriority	1	Lowest possible value.
subbranchPriority	5	Feature and refactor branches are five times more likely to get a new commit than the main branch.
initialGraph	Empty Graph	The history will generate a new graph from scratch.

Table 5.1: The default values of the history generator configuration.

## 6 Evaluation

To quantitatively evaluate our algorithm, we use the values of the survey by Zou et al. [Zou+19] as a reference point. They found the general size of their history graphs to have a mean of 918.3 commits with a maximum of 25,052 commits. For branches, on the other hand, 87.9% only had five or less, while only 3.1% had more than 30. All tests were performed on a ThinkPad E595 with an AMD Ryzen 7 3700U CPU and 16 GB of DDR4-SDRAM.

### 6.1 History generation

First we want to quantify how long it takes to generate the empty history. This allows us to address RQ1 and RQ1a. To make sure we can match realistic values, we generate a history graph that is bigger than the ones of the survey by Zou et al. [Zou+19]. Since the history generation is random, we choose to repeat the tests for 300 different seeds to gain an average value. We choose to vary the amount of commits, the average number of parallel branches, and the length of each branch as the three key values we want to evaluate, because they determine the shape and size of the history graph. This also allows us to use the `createSimpleConfiguration` factory method to generate all configurations for these test runs.

For the amount of commits, we chose 100,000 to 500,000 commits. This is a lot bigger than the maximum of 25,000 commits from the study by Zou et al. [Zou+19]. We split this range into 9 evaluation runs with a 50,000-commit difference between each. We assume that there is a linear dependency between the total amount of commits and the generation runtime. This is because each branch only looks at its most recent commit and ignores the history graph otherwise.

For the average amount of parallel branches, we chose 100 to 1,000. Zou et al. [Zou+19] only quantified the total amount of branches and cut the metric at 30+. So we chose a value at least double of that as the minimum value. We split this range into 10 runs with 100 branches as the step size between each run. We expect that there is a linear dependency between the generation runtime and the average amount of branches, because the branch selection logic will need more time the more branches there are.

For the length of the branches, we chose 100 to 1,000 commits. Since Zou et al. [Zou+19] did not have any metrics on the branch length, we selected this value in relation to the total amount of commits. We also split this range into 10 runs with a step size of 100 commits between each run. We expect that an increase in branch length will slightly reduce the generation runtime because overall fewer branch objects will be created and destroyed.

We varied one parameter at a time choosing the minimum values of the respective ranges for the parameters that are not varied. Table 6.1 shows all values of the configuration.

Parameter	Value
seed	0 - 299
amountCommits	100,000 - 500,000
branchLengthSequence	Normal distribution around a mean of 100 - 1,000
commitSizeSequence	Uniform distribution between 2 and 30
nameGeneratorSequence	UUID Strings
mainGraphGenerator	LegacyGeneratorAdapter with default configuration
featureGraphGenerator	
refactorGraphGenerator	
graphMerger	RandomRejectMerger
maxParallelBranches	medianParallelBranches + 40%
minParallelBranches	medianParallelBranches - 40%
medianParallelBranches	100 - 1,000
parallelBranchesSpread	0.7
subbranchSequence	RandomBooleanSequence 10% True
refactorBranchSequence	RandomBooleanSequence 30% Refactors
deadBranchSequence	RandomBooleanSequence 10% True
mainBranchPriority	1
subbranchPriority	5
initialGraph	Empty Graph

Table 6.1: Configuration values of the test runs when measuring the history generation. The graph generators, the merger, and the initial graph are not used, as only the history generation step is measured.

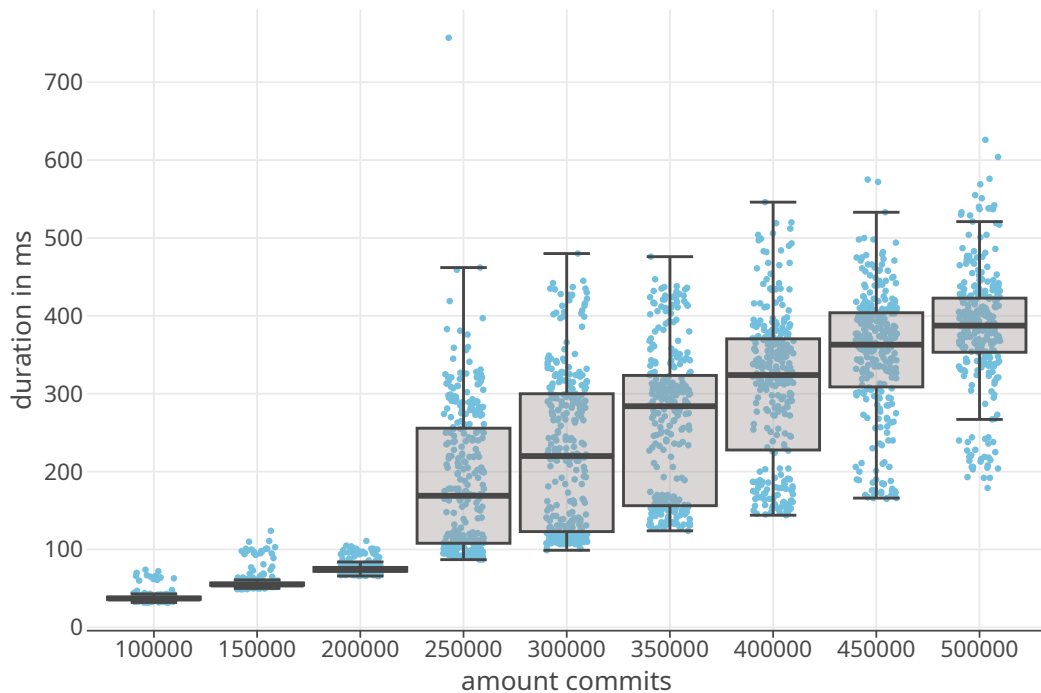


Figure 6.1: The generation runtime for histories with different amounts of commits. The amount of commits varies between 100,000 and 500,000 commits, the amount of parallel branches is fixed at 100 branches, and the branch length is fixed at 100 commits. The other parameter can be found in Table 6.1.



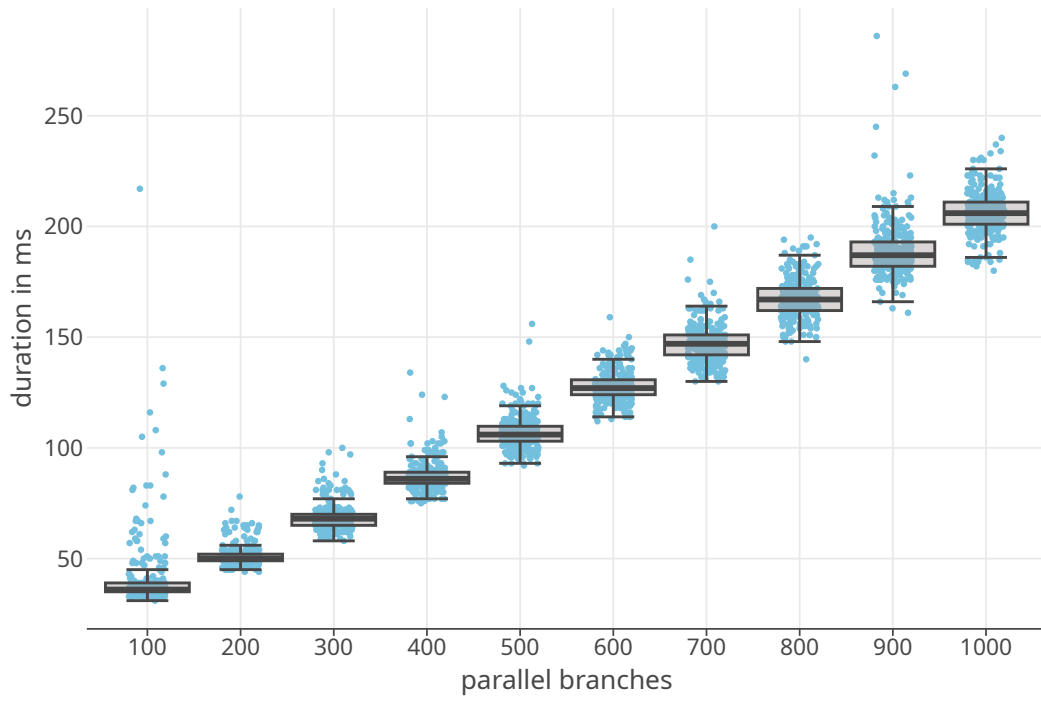


Figure 6.2: The generation runtime for histories with different amounts of parallel branches. The amount of parallel branches varies between 100 and 1,000 branches, the amount of total commits is fixed at 100,000 commits, and the branch length is fixed at 100 commits. The other parameter can be found in Table 6.1.

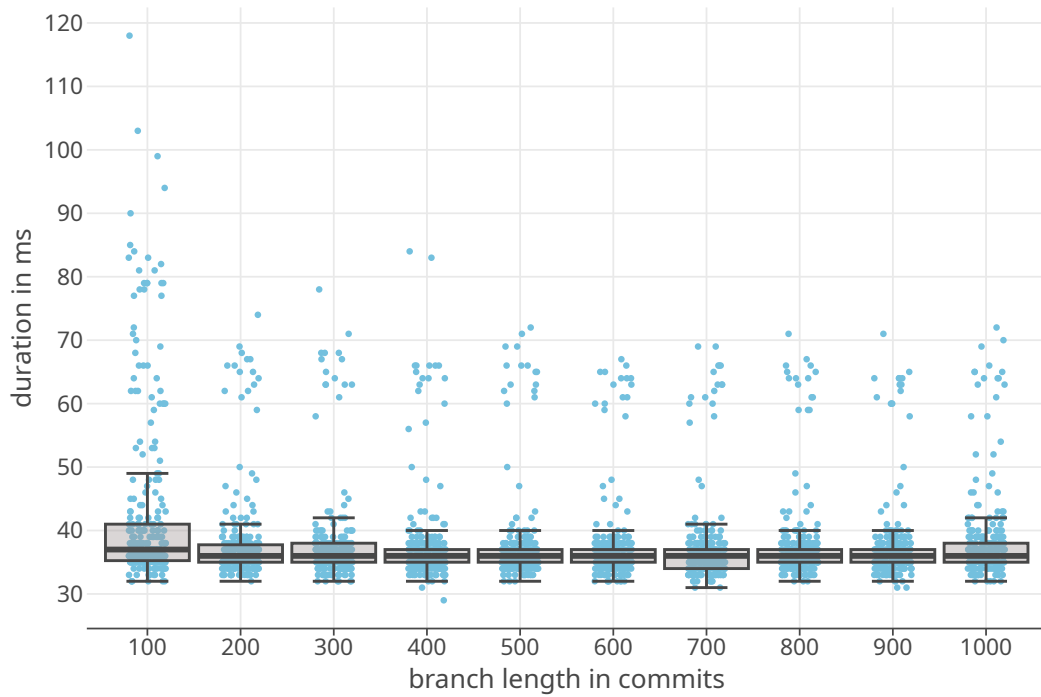


Figure 6.3: The generation runtime for histories with different branch lengths. The branch length varies between 100 and 1,000 commits, the amount of parallel branches is fixed at 100 branches, and the amount of total commits is fixed at 100,000 commits. The other parameter can be found in Table 6.1.

Figure 6.1 shows the generation time when varying the total amount of commits. As expected, we see a cluster of runs that grows proportional to the amount of commits. Starting with 250,000, a second cluster emerges. The second cluster also seems to grow proportional to the amount of commits. We assume this is due to hardware limitations, but to explore that hypothesis more tests on different hardware and a wider range of total commits are needed.

Figure 6.2 shows the generation time when varying the amount of parallel branches. Here we can see a clear proportional relation between the parallel branches and the average runtime duration.

Figure 6.3 shows the generation time when varying the branch length. While there is a small decrease in runtime duration when moving from 100 to 200 commits in branch length, there is no visible dependency between the branch length and the runtime duration for the other values. We assume that we overestimated the impact of the branch object creation and destruction on the overall runtime duration.

## 6.2 History population

To answer our research questions, we also need to measure the runtime of the history population. Since the runtime of the graph generators also depends on the model size, we chose to supply a precomputed initial graph with a total of 1,000 nodes and edges and tune the generators to generate commits that are mostly size neutral. The overall runtime of the graph generators is considerably longer than the runtime of the history generation. To compensate for the limited resources we had to reduce parameters for the history generation. We reduced the amount of different seeds to 30. For the amount of commits, we chose 1,000 as the minimum value, which is close to the mean of 918 from the survey of Zou et al. [Zou+19], and 10,000 as the upper limit. While that is smaller than the maximum of 25,052, it is still six standard deviations (1,483) above the mean. To accommodate the smaller history size, we reduce the number of parallel branches to 20. This is well above 87.9% of the sample projects,

Parameter	Value
seed	0 - 29
amountCommits	1,000 - 10,000
branchLengthSequence	Normal distribution around a mean of 100
commitSizeSequence	Uniform distribution between 2 and 30
nameGeneratorSequence	UUID Strings
mainGraphGenerator	LegacyGeneratorAdapter or CircleGeneratorAdapter
featureGraphGenerator	
refactorGraphGenerator	
graphMerger	RandomRejectMerger
maxParallelBranches	28
minParallelBranches	12
medianParallelBranches	20
parallelBranchesSpread	0.7
subbranchSequence	RandomBooleanSequence 10% True
refactorBranchSequence	RandomBooleanSequence 30% Refactors
deadBranchSequence	RandomBooleanSequence 10% True
mainBranchPriority	1
subbranchPriority	5
initialGraph	precomputed graph with ~1,000 nodes + edges

Table 6.2: Configuration values of the test runs when measuring the history population.

which only had five branches. Table 6.2 shows the configuration parameters for the history population test runs.

For the circle generator we generate an initial graph by running one generation step with 1,000 atomic actions and a removal chance of 0%. Since the circle generator cannot move or rename nodes, the resulting graph will have a total of 1,000 nodes and edges. The generator we supply to the configuration has a removal chance of 50%, making it size neutral. Because the circle generator supports nested nodes but does not count them when deleting, we set the chance that new circles are spawned anywhere but the root graph to 0%. This is needed to ensure size neutrality.

For the GraphGentool we use its graph factory to generate the initial graph with the default values provided by the factory and a size of 1,000. To create a size-neutral graph generator, we supply the LegacyGeneratorAdapter with the edit chances of 15% of adding a simple node, 5% of adding a region node, 20% of deleting any node, 15% of moving a node, 5% of changing the color of a simple node, 20% of adding an edge, and 20% of deleting an edge. The atomic edit parameter is set to true by default.

We expect that the model size for both generators will stay around 1,000 commits, while the population runtime increases proportionally to the amount of commits.

### 6.2.1 Circle Graph Generator

Figure 6.4 shows the population runtime of when using the circle generator for increasing amounts of total commits. From 1,000 to 7,000 commits, we see the generation runtime rise proportional to the amount of commits. Additionally, we see that the spread between different seeds increases with increasing amounts of commits. This is to be expected as the additions and deletions are random. After 7,000 commits the generation runtime stagnates. This seems anomalous, as generating more graph models should increase the runtime. We assume this is due to the increasing spread in runtime duration and the small amount of seeds. But since the model graph generation runtime also depends on the model size, we also have a look at

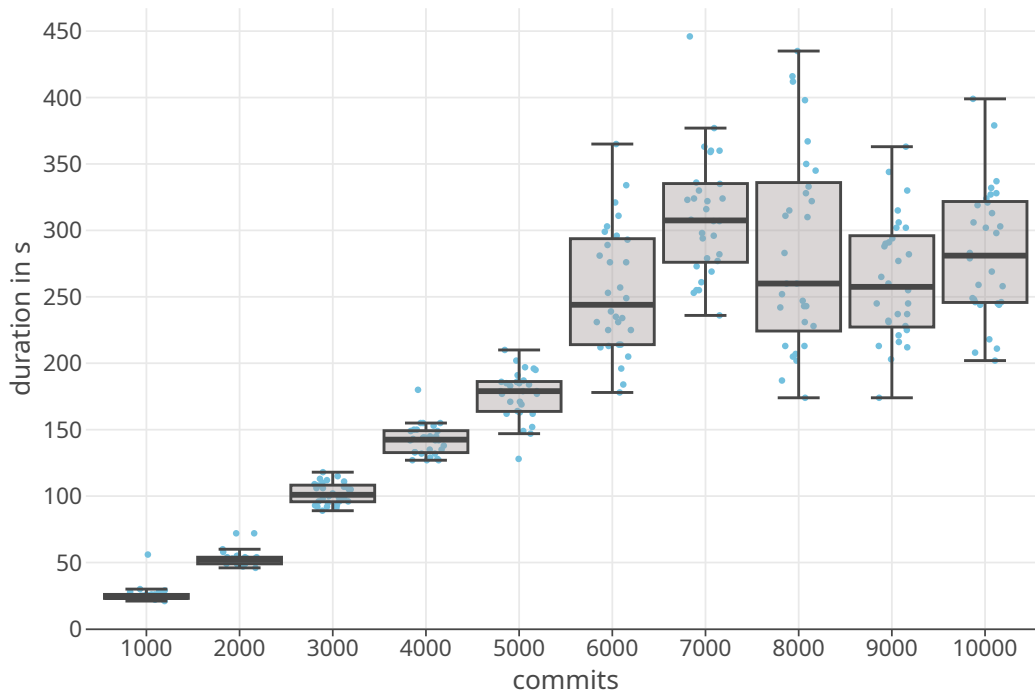


Figure 6.4: The generation runtime for the history population in relation to the total amount of commits using the circle generator and the configuration from Table 6.2.

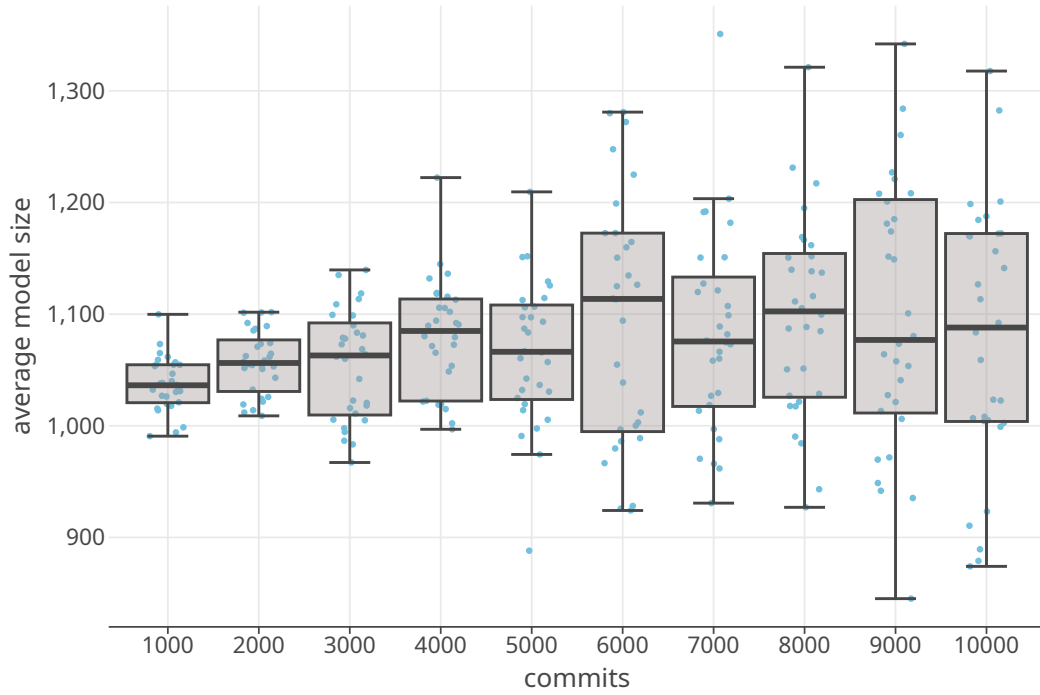


Figure 6.5: The average model graph size as the sum of edges and nodes in relation to the total amount of commits, generated with the circle generator using the configuration from Table 6.2. The average is collected as the median over all commits of the history graph of each run.

commits	min nodes	min edges	average nodes	average edges	max nodes	max edges
1000	356.5	355.7	519.1	518.2	696.2	695.3
2000	314.1	313.2	528.8	527.9	769.6	768.6
3000	286.3	285.6	527.5	526.5	778.5	777.4
4000	246.3	245.5	540.0	539.0	861.1	859.9
5000	203.7	202.9	533.4	532.4	903.9	902.7
6000	196.8	196.1	547.8	546.8	948.1	947.1
7000	169.6	168.6	541.8	540.8	925.1	924.2
8000	164.5	163.6	549.3	548.4	950.4	949.5
9000	132.9	132.2	549.5	548.6	993.6	992.5
10000	115.7	114.7	540.1	539.1	962.1	961.0

Figure 6.6: The variety in model graph split into edges and nodes in relation to the total amount of commits, generated with the circle generator using the configuration from Table 6.2. The values are collected as the median of the 30 different runs with the same amount of commits.

the generated model sizes. Figure 6.5 charts the average model size as the sum of the nodes and edges in relation to the amount of commits. The average is collected over all commits of the generated history graph. This means strong size differences in a singular commit do not have a huge impact on the average. Still a trend of a stable model size with an increasing spread is clearly visible. Figure 6.6 shows the size variation inside each history graph. The values are collected as averages over all 30 different seeds. Due to the structure of the circle graphs, the amount of edges and nodes are almost identical. With an increasing amount of commits, we see that the variety in model size also increases. At 10,000 commits, we see that on average the minimum model size during the generation is about one fifth of the average and

the maximum is almost double the average. This could be a reason for the falling generation runtime durations for a high amount of commits. Further tests on the relation between the runtime of the circle generator and the model size are needed to draw definitive conclusions.

### 6.2.2 GraphGentool

Figure 6.7 shows the population runtime of when using the GraphGentool for increasing amounts of total commits. Similar to the circle graph generator, we can see the average runtime rise proportional to the amount of commits. Here the trend continues up to the 10,000 commits. We can also see the rise in the spread of runtime duration. In contrast to the circle generator, we can see a second trend, which is the exponential growth of the maximum runtime duration. In Figure 6.8 we see the reason for this growth. It plots the average model size as a sum of edges and nodes against the total amount of commits. Here we can see that while the average value stays close to the initial 1,000, the maximum value grows proportional to the amount of commits. This means for more commits, the outliers have progressively bigger models, leading to an exponential growth in runtime duration. Figure 6.9 shows again the variety of model graph sizes inside each history graph. In contrast to the circle graph generator, we see a discrepancy between the amount of nodes and edges. Even though the provided chances to add an edge and add a region or simple node are equal, we can see that with more commits, the maximum amount of nodes rises faster than the maximum amount of edges. Additionally, we see that the minimum amount of edges and nodes stays constant. This indicates that the model graphs are growing more than they are shrinking and therefore have the minimum values at the beginning of the history graph. A reason for that can be the `atomicEdit` parameter. As already mentioned in section 5.4.1, having this parameter on skews chances away from edits with additional implicated edits. Those edits are node deletions. Since edges always need a node, deleting a connected node implicates the deletion of the edge as well. Additionally, when deleting a region, all nodes and edges in the region's graph are implicated as well. When the variant generator of the GraphGentool finds that the chosen edit has too many implications

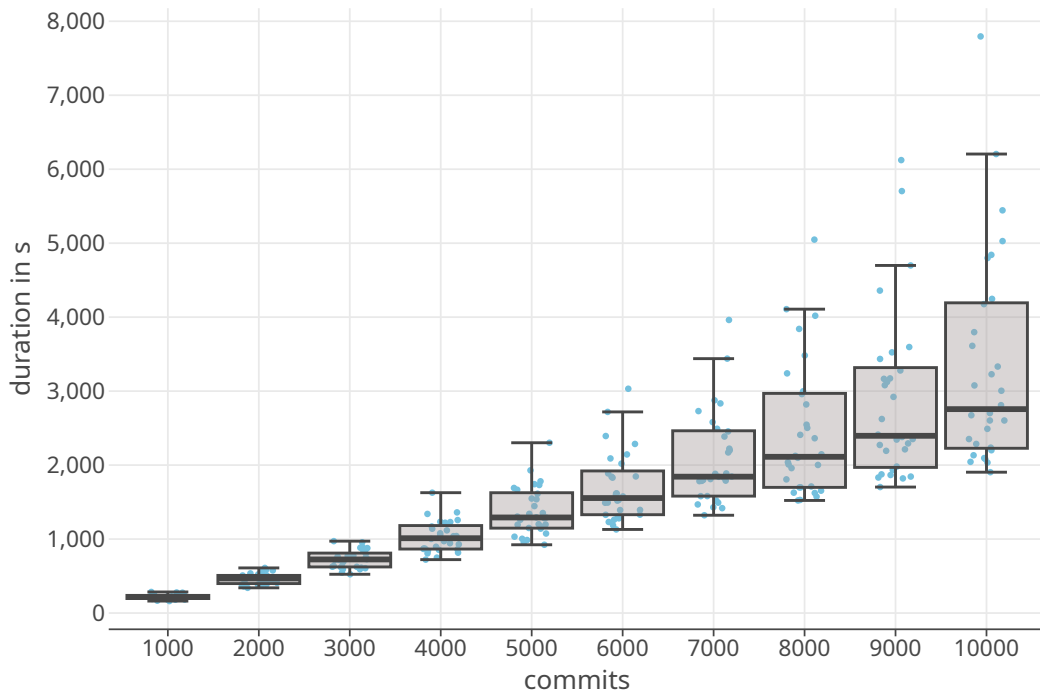


Figure 6.7: The generation runtime for the history population in relation to the total amount of commits using the GraphGentool and the configuration from Table 6.2.

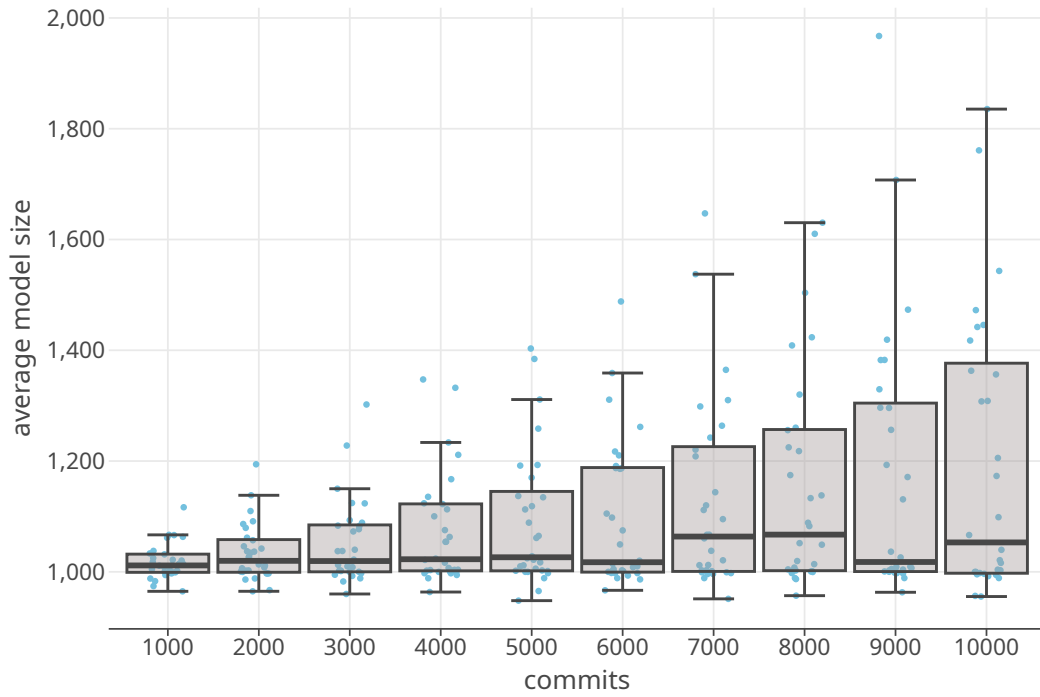


Figure 6.8: The average model graph size as the sum of edges and nodes in relation to the total amount of commits, generated with the GraphGentool using the configuration from Table 6.2. The average is collected as the median over all commits of the history graph of each run.

commits	min nodes	min edges	average nodes	average edges	max nodes	max edges
1000	491.9	480.0	519.6	498.0	622.2	533.3
2000	491.9	479.7	533.2	500.5	692.2	552.5
3000	491.9	479.7	547.0	504.0	757.5	571.3
4000	491.9	479.7	564.6	507.4	818.5	583.9
5000	491.9	479.7	580.0	508.5	871.5	588.6
6000	491.9	479.7	585.2	512.1	891.4	600.0
7000	491.7	479.7	611.0	514.9	990.1	613.6
8000	491.7	479.7	630.4	521.4	972.4	624.0
9000	491.9	479.7	644.1	524.4	1044.3	630.7
10000	491.9	479.7	662.9	529.0	1090.2	653.0

Figure 6.9: The variety in model graph split into edges and nodes in relation to the total amount of commits, generated with the GraphGentool using the configuration from Table 6.2. The values are collected as the median of the 30 different runs with the same amount of commits.

for the desired amount of edits, a different edit is chosen. For huge amounts of commits, this effect can compound, leading to nested nodes that are unlikely to ever be deleted because the amount of implicated edits is too big. Because these skipped edits are deletes, this also explains the asymmetric growth of the outliers in Figure 6.9 and 6.8.

### 6.2.3 Merges

To completely answer RQ1a, we also have to take into consideration that there is an added time for merge generation. In our experimental setup we used the RandomRejectMerger as the

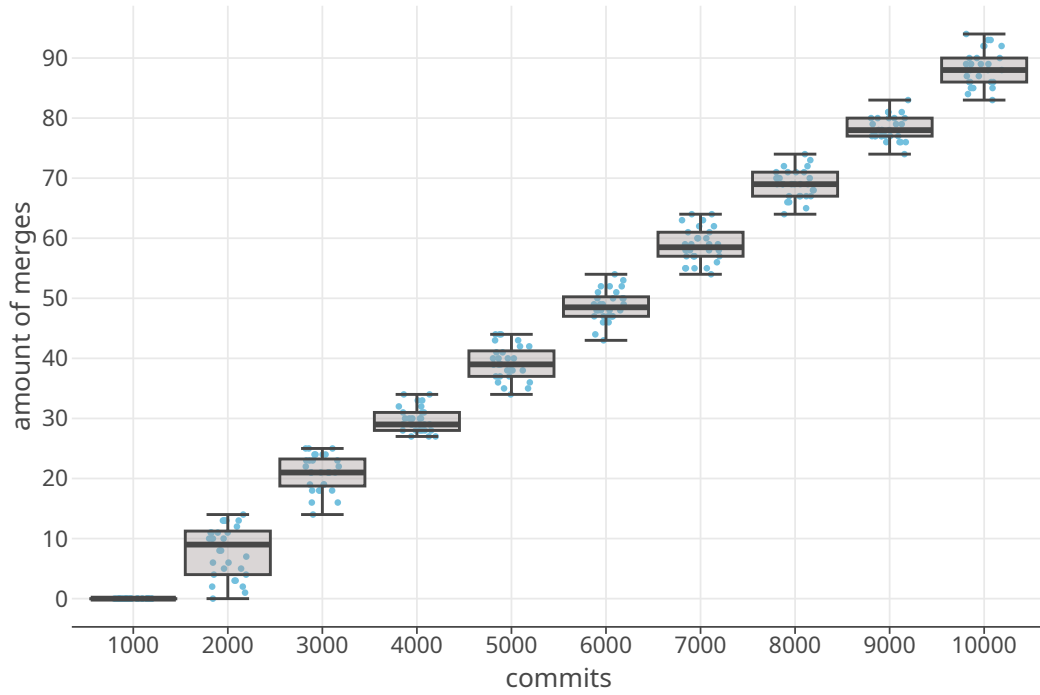


Figure 6.10: The amount of merges during the history population when using the parameters from Table 6.2.

replacement for a real merger. The runtime of this merger is very low, as it randomly returns a copy of one of the input branches. A real merger that is computing the difference between both graphs will most likely have a considerably longer runtime. To get a grasp on the impact a merger would have on the population runtime, we monitored the amount of merges for the different amounts of commits. Figure 6.10 shows the clear proportional relation between the amount of commits and the amount of merges. This is to be expected. When comparing the amount of merges with population durations from Figure 6.7, we see that for 10,000 commits, the population time is around 2,800 seconds or 47 minutes, while there are around 90 merges. This would mean a merger would have to calculate one merge in 0.3 seconds or less in order to stay two orders of magnitudes below the linear history. Figure 6.8 shows that the model size of the model graphs that need to be merged is just slightly above 1,000 nodes and edges.

### 6.3 Research Questions

In the following section we will address our research questions again and try to answer them using our findings.

#### RQ1: Can we improve upon the existing generators to generate a realistic development history?

As shown earlier in the evaluation, we are able to generate and populate a history that matches or surpasses the median values found in the survey by Zou et al. [Zou+19]. Using the adapter described in Section 5.4.1, we use the existing capabilities of the GraphGentool and repurpose the ability to generate variants to generate commits. This is a direct improvement to the linear variants of the GraphGentool.

**RQ1a: Can we generate the branched history without an unreasonable increase in compute time compared to a linear one?**

Comparing the values of Figure 6.1 and 6.7, we can see that for 10,000 commits, the population takes 2800 seconds, while the generation of a history graph with 10 times more commits only takes 0.04 seconds. This is way below the target of being two orders of magnitude lower than the population. But the history graph generation is not the only thing that differentiates the linear history from a branched one. We also have to take the merges into consideration. In the previous section we found that we need a merger that is able to merge two graphs, each having a sum of 1,000 nodes and edges, in 0.3 seconds or less to stay two magnitudes of power below the generation time of a linear history.

**RQ1b: What is the runtime to generate the MDSE model graphs for a realistic history?**

Due to the wide variety in history graphs, the generation runtime will vary similarly. Figure 6.7 shows that generating a history with 1,000 commits takes around 250 seconds. This matches the median size of the projects from the study by Zou et al. [Zou+19]. With increasing history size, the variety in runtime duration increases as well. We assume this is due to the variety in model graph sizes with increasing amounts of commits. When comparing the GrapGentool in Figure 6.7 to the circle generator in Figure 6.4, we see that the population runtime duration majorly depends on the used generator.

**RQ2: How does a modular design look like that allows to extend the history generation with different MDSE model graph generators?**

In Section 5.4 we describe the adapters we use to add different graph generators to the history generation. By switching the adapter in the configuration we pass to the HistoryBuilder class, we specify which generators are used during the history population.



## 7 Conclusion

In this work we presented a novel algorithm that generates a branched history graph and uses the GraphGentool to populate it. The algorithm is highly configurable using the RandomSequence we introduced to allow for different probability distributions for its random aspects. We measured the GraphGentools population time of a realistically sized history of 1,000 commits with model graphs with a size of 1,000 nodes and edges combined to be around 4 minutes and showed that the computational overhead of generating the history graph is negligible. Additionally, we introduced the circle generator, proving the ease of extensibility of our algorithm. Overall we demonstrated that our algorithm is efficient and adaptable, providing a starting point for future extensions and integrations into large-scale model generation projects.

Future works could explore the impact a merger has on the population time. Additionally, our algorithm uses the GraphGentools graph metamodel for its model graph. This is not necessary and could be generalized by making the commits and generator adapter generically typed. There is also the possibility to increase the parameterization of the algorithm by extending the RandomSequence class to make conditional randoms. Right now each random sequence produces a new random value based on the parameters passed during initialization. But random values can also depend on the current state of the generation. Mapping those onto easy-to-understand random sequences could allow for more expressive configurations.

# List of Figures

1.1	History Graph GraphGentool . . . . .	7
2.1	Simplified pipeline of a model translational framework . . . . .	11
2.2	Software Lifecycle . . . . .	13
2.3	General Fuzzing Process . . . . .	15
2.4	Data Generation Using Generative Models . . . . .	16
2.5	Shapes of version graphs . . . . .	17
2.6	Purposes of Branches . . . . .	17
2.7	GitFlow . . . . .	18
2.8	Graph Generator Classification . . . . .	19
3.1	Exploration Process Model Genration . . . . .	23
4.1	UML Class Diagram Model Graph . . . . .	24
4.2	UML Class Diagram Graph Delta . . . . .	25
4.3	UML Class Diagram History . . . . .	26
4.4	Sample History Graph Generation . . . . .	27
5.1	History Generation Workflow . . . . .	29
5.2	Pseudocode History Generation . . . . .	30
5.3	Sample Branch Generation . . . . .	32
5.4	Motif Generation Circle Graph . . . . .	35
6.1	Generation runtime when varying total amount of commits . . . . .	40
6.2	Generation runtime when varying the amount of parallel branches . . . . .	41
6.3	Generation runtime when varying branch length . . . . .	41
6.4	Generation Runtime Circle Generator . . . . .	43
6.5	Average Model Size Circle Generator . . . . .	44
6.6	Variety in Model Size Circle Generator . . . . .	44
6.7	Generation Runtime GraphGentool . . . . .	45
6.8	Average Model Size GraphGentool . . . . .	46
6.9	Variety in Model Size GraphGentool . . . . .	46
6.10	Amount of Merges by Total Commits . . . . .	47

# Bibliography

- [AZW06] Uwe Aßmann, Steffen Zschaler, and Gerd Wagner. “Ontologies, meta-models, and the model-driven paradigm”. In: *Ontologies for software engineering and software technology*. Springer, 2006, pp. 249–273.
- [Ben00] Keith Bennett. “Software Maintenance and Evolution: a Roadmap”. In: *Proceedings of the Conference on The Future of Software Engineering*. May 2000, pp. 73–87. DOI: [10.1145/336512.336534](https://doi.org/10.1145/336512.336534).
- [Bon+20] Angela Bonifati et al. “Graph Generators: State of the Art and Open Challenges”. In: *ACM Comput. Surv.* 53.2 (Apr. 2020). ISSN: 0360-0300. DOI: [10.1145/3379445](https://doi.org/10.1145/3379445). URL: <https://doi.org/10.1145/3379445>.
- [Bro+19] Marc Brockschmidt et al. *Generative Code Modeling with Graphs*. 2019. arXiv: [1805.08490](https://arxiv.org/abs/1805.08490) [cs.LG]. URL: <https://arxiv.org/abs/1805.08490>.
- [CEE22] Julio César Cortés Ríos, Suzanne M Embury, and Sukru Eraslan. “A unifying framework for the systematic analysis of Git workflows”. en. In: *Inf. Softw. Technol.* 145.106811 (May 2022), p. 106811.
- [Che+25] Boqi Chen et al. “Accurate and Consistent Graph Model Generation from Text with Large Language Models”. In: *2025 ACM/IEEE 28th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE. 2025, pp. 130–141.
- [CM13] Catarina Costa and Leonardo Murta. “Version Control in Distributed Software Development: A Systematic Mapping Study”. In: *2013 IEEE 8th International Conference on Global Software Engineering*. 2013, pp. 90–99. DOI: [10.1109/ICGSE.2013.19](https://doi.org/10.1109/ICGSE.2013.19).
- [CS14] Scott Chacon and Ben Straub. *Pro git*. Springer Nature, 2014.
- [CW98] Reidar Conradi and Bernhard Westfechtel. “Version models for software configuration management”. In: *ACM Comput. Surv.* 30.2 (June 1998), pp. 232–282. ISSN: 0360-0300. DOI: [10.1145/280277.280280](https://doi.org/10.1145/280277.280280). URL: <https://doi.org/10.1145/280277.280280>.
- [DD25] Kyanna Dagenais and Istvan David. “Complex Model Transformations by Reinforcement Learning with Uncertain Human Guidance”. In: *2025 ACM/IEEE 28th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2025, pp. 209–220. DOI: [10.1109/MODELS67397.2025.00025](https://doi.org/10.1109/MODELS67397.2025.00025).
- [Dri10] Vincent Driessen. “A successful Git branching model”. In: URL <http://nvie.com/posts/a-successful-git-branching-model> (2010).

- [FK96] Roger Ferguson and Bogdan Korel. "The chaining approach for software test data generation". In: *ACM Trans. Softw. Eng. Methodol.* 5.1 (Jan. 1996), pp. 63–86. ISSN: 1049-331X. DOI: [10.1145/226155.226158](https://doi.org/10.1145/226155.226158). URL: <https://doi.org/10.1145/226155.226158>.
- [GZ23] Xiaojie Guo and Liang Zhao. "A Systematic Survey on Deep Generative Models for Graph Generation". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45.5 (2023), pp. 5370–5390. DOI: [10.1109/TPAMI.2022.3214832](https://doi.org/10.1109/TPAMI.2022.3214832).
- [Jää16] Esa Jääskelä. "Genetic algorithm in code coverage guided fuzz testing". MA thesis. E. Jääskelä, 2016.
- [Keg+24] Karl Kegel et al. "A Variance-Based Drift Metric for Inconsistency Estimation in Model Variant Sets." In: *J. Object Technol.* 23.3 (2024), pp. 1–14.
- [LCC22] José Antonio Hernández López, Javier Luis Canovas Izquierdo, and Jesús Sánchez Cuadrado. "ModelSet: a dataset for machine learning in model-driven engineering". In: *Software and Systems Modeling* 21.3 (2022), pp. 967–986.
- [Lia+18] Hongliang Liang et al. "Fuzzing: State of the Art". In: *IEEE Transactions on Reliability* 67.3 (2018), pp. 1199–1218. DOI: [10.1109/TR.2018.2834476](https://doi.org/10.1109/TR.2018.2834476).
- [Lu+25] Yingzhou Lu et al. *Machine Learning for Synthetic Data Generation: A Review*. 2025. arXiv: [2302.04062](https://arxiv.org/abs/2302.04062) [cs.LG]. URL: <https://arxiv.org/abs/2302.04062>.
- [McM04] Phil McMinn. "Search-based software test data generation: a survey". In: *Software Testing, Verification and Reliability* 14.2 (2004), pp. 105–156. DOI: <https://doi.org/10.1002/stvr.294>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.294>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.294>.
- [Moh25] Partha Sarathi Mohapatra. "Artificial Intelligence-Driven Test Case Generation in Software Development". In: *Intelligent Assurance: Artificial Intelligence-Powered Software Testing in the Modern Development Lifecycle* 4 (2025), p. 38.
- [MV06] Tom Mens and Pieter Van Gorp. "A taxonomy of model transformation". In: *Electronic notes in theoretical computer science* 152 (2006), pp. 125–142.
- [Nas+20] Nebras Nassar et al. "Generating Large EMF Models Efficiently: A Rule-Based, Configurable Approach". In: *Fundamental Approaches to Software Engineering: 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings*. Dublin, Ireland: Springer-Verlag, 2020, pp. 224–244. ISBN: 978-3-030-45233-9. DOI: [10.1007/978-3-030-45234-6\\_11](https://doi.org/10.1007/978-3-030-45234-6_11). URL: [https://doi.org/10.1007/978-3-030-45234-6\\_11](https://doi.org/10.1007/978-3-030-45234-6_11).
- [Pid09] Michael Pidd. *Tools for thinking modelling in management science Michael Pidd*. 3rd ed. Wiley, 2009.
- [RA23] Guru Pramod Rusum and Sunil Anasuri. "Synthetic Test Data Generation Using Generative Models". In: *International Journal of Emerging Trends in Computer Science and Information Technology* 4.4 (2023), pp. 96–108.
- [Raj14] Václav Rajlich. "Software evolution and maintenance". In: *Future of Software Engineering Proceedings*. FOSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 133–144. ISBN: 9781450328654. DOI: [10.1145/2593882.2593893](https://doi.org/10.1145/2593882.2593893). URL: <https://doi.org/10.1145/2593882.2593893>.

- [RS16] N. Rama Rao and K. Chandra Sekharaiah. "A Methodological Review Based Version Control System with Evolutionary Research for Software Processes". In: *Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies*. ICTCS '16. Udaipur, India: Association for Computing Machinery, 2016. ISBN: 9781450339629. DOI: [10.1145/2905055.2905072](https://doi.org/10.1145/2905055.2905072). URL: <https://doi.org/10.1145/2905055.2905072>.
- [Sch+06] Douglas C Schmidt et al. "Model-driven engineering". In: *Computer-IEEE Computer Society*- 39.2 (2006), p. 25.
- [Sei03] Ed Seidewitz. "What Models Mean". In: *IEEE Softw.* 20.5 (Sept. 2003), pp. 26–32. ISSN: 0740-7459. DOI: [10.1109/MS.2003.1231147](https://doi.org/10.1109/MS.2003.1231147). URL: <https://doi.org/10.1109/MS.2003.1231147>.
- [Sem+21] Oszkár Semeráth et al. "Automated generation of consistent, diverse and structurally realistic graph models". In: *Software and Systems Modeling* 20.5 (2021), pp. 1713–1734.
- [Ver+25] Charlotte Verbruggen et al. "Toward a Community-Curated Golden Dataset of UML Models". In: *2025 ACM/IEEE 28th International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 2025, pp. 43–50. DOI: [10.1109/MODELS-C68889.2025.00012](https://doi.org/10.1109/MODELS-C68889.2025.00012).
- [VW07] A van Deursen, E Visser, and J Warmer. "Model-Driven Software Evolution: A Research Agenda". English. In: *Proceedings 1st International Workshop on Model-Driven Software Evolution (MoDSE)*. Ed. by Dalila Tamzalit. University of Nantes, 2007, pp. 41–49.
- [Xia+22] Sheng Xiang et al. "General graph generators: experiments, analyses, and improvements". en. In: *VLDB J.* 31.5 (Sept. 2022), pp. 897–925.
- [Zho+18] Zhenpeng Zhou et al. "Optimization of Molecules via Deep Reinforcement Learning". In: *CoRR* abs/1810.08678 (2018). arXiv: [1810.08678](https://arxiv.org/abs/1810.08678). URL: <http://arxiv.org/abs/1810.08678>.
- [Zou+19] Weiqin Zou et al. "Branch Use in Practice: A Large-Scale Empirical Study of 2,923 Projects on GitHub". In: *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*. 2019, pp. 306–317. DOI: [10.1109/QRS.2019.00047](https://doi.org/10.1109/QRS.2019.00047).