





A Conceptual Framework for Fine-Grained Quality Assessment in Version Graphs


Karl Kegel 
Technische Universität Dresden
Dresden, Germany
karl.kegel@tu-dresden.de

Romain Pascual 
Centralesupélec, Université Paris-Saclay
Gif-sur-Yvette, France
romain.pascual@centralesupelec.fr

Kevin Feichtinger 
Karlsruhe Institute of Technology
Karlsruhe, Germany
kevin.feichtinger@kit.edu

Andreas Domanowski 
Technische Universität Dresden
Dresden, Germany
andreas.domanowski@tu-dresden.de

Marie Clausnitzer 
Technische Universität Dresden
Dresden, Germany
marie.clausnitzer@mailbox.tu-dresden.de

Uwe Aßmann 
Technische Universität Dresden
Dresden, Germany
uwe.assmann@tu-dresden.de

Abstract—Today’s software development and modeling processes happen collaboratively. A key challenge in collaborative software engineering is assessing the quality properties of the different system revisions developed by different people. Modern solutions for version control, variation control, and model management can track and relate versions, feature variations, branches, and experimental modifications. We particularly observe a trend in the development of model management tools to support increasingly fine-grained evolution operations and artifact relationships. This detailed knowledge of a system’s revision history enables history-based quality assessment. However, the current state of the art lacks a comprehensive terminology framework for clearly describing the scope within which a particular quality attribute is assessed in the revision history. This leads to inefficient communication and hinders tackling the actual objective of the quality assessment. This work contributes a formal framework for quality regions in revision management systems. Our framework is based on revision graphs and incorporates concepts such as versions, variants, merges, multi-systems, and views, building a common ground for fine-grained quality assessment of evolving systems. We provide an open-source library implementation of our formal concepts, which can be used to extend existing version control or model management tools.

Index Terms—software engineering, quality assurance, technical debt, versioning, model management

I. INTRODUCTION

Software engineers work collaboratively on artifacts such as models or code. However, collaboration does not necessarily mean that people sit together and work jointly on a single artifact. *Version control systems (VCS)* [1] enable the management of these workflows. A prominent example of a VCS is *Git*, which is typically used together with a collaboration platform such as *GitHub* or *GitLab*. Despite the prominence of *Git*, many different domain- and purpose-specific versioning systems exist. Examples are found in modeling and software product-line engineering [2]–[4]. On an abstract level, these management tools have similar functionalities and purposes. Therefore, we remain, for large parts of this work, on a conceptual level, not distinguishing between modeling in particular and general software engineering. However, we see

the largest impact of this work in the field of model-driven software engineering, particularly in model management and model evolution. The reason is the focus of modern model management tools on increasingly fine-grained support for evolution processes beyond commits and branches, for example, the support of the co-evolution of linked models, the integrated handling of views [4], or the integration of variability management into version control [3].

Before continuing with the further presentation of this work’s context and problem statement, we provide a brief introduction to the terminology used. Software engineers often use the terms’ *version*, *variant*, *revision*, or *commit* ambiguously and interchangeably. We are aware of the terms’ definitions in the literature [1], [5]–[7], which we discuss in Sec. II. On this work’s abstract level, we prefer to use the term *revision* as a unified descriptor of the “things that are managed” [7]. A revision is a (development-)state of a system identified by its successors and predecessors in time. Subsequently, we refer to any system that manages revisions as a *revision management system (RMS)*. We particularly allow an RMS to comprise multiple revision histories, i.e., support for individual versioning at an arbitrary granularity, as illustrated in Section I-E. This work’s notion of an RMS should not be confused with a *feature model* [8] or *hyper-feature model* [9]. Our notion of an RMS is not meant to represent a system’s configuration space in terms of a software product line. However, we acknowledge that the distinction between RMS and *variation control systems* [2] for software product lines can be fluid.

A. Context

A key challenge in software and systems engineering is ensuring the desired quality properties of a system during its development as motivated by standards such as ISO 25010 [10] and ISO 25023 [11]. If neglected, *technical debt* builds up over time [12], [13]. Although quality assurance and measuring different quality attributes are well researched, existing works and tools focus on analyzing a single system revision. It

remains unclear, how the quality of a network of revisions can be assessed. For instance, a prominent quality property that can be measured over multiple revisions is *consistency*. Examples for consistency problems are breaking changes in a sequence of updates, non-mergeable changes implemented in two branches, or parallel developed features that turn out to be incompatible. If one now defines a set of metrics to measure and track the consistency violations - or any other metric -, one must specify the location within the revision history where the metrics operates. One could assess the metric along the time axis, i.e., comparing a specific revision and its k most recent predecessors. One could assess the metric along the space axis, i.e., comparing a specific revision and its k closely related variants. We call these scopes *quality regions*. To assess quality properties within a system comprising multiple revisions, it is essential to clearly define and communicate the region within a revision history a quality assessment is applied to.

B. Problem Statement

Quality assurance of versioned systems is a field with a large research potential. In this field, we identify the absence of a nomenclature and formal framework of regions for quality assessment in revision histories as main obstacles towards a clear definition and communication of quality properties. Working with complex networks of revisions, it must be clear over which subsets of revisions a quality property should hold. We, therefore, ask the research question “**How can we constructively define a general framework of quality regions in revision management based on a suitable formal abstraction of revision management systems?**” By answering this question, we want to enable future works to perform time- and space-aware quality analysis, e.g., the calculation of different metrics or the optimization of version histories on a concise formal and technical foundation.

C. Approach

We aim to address the research question of this work in two stages. First, we propose a formal representation for generally describing revision structures in a revision management system. We base our formalism on the well-known structure of *version graphs* [1], [5], [6]. Second, we define the actual formal framework of quality regions. To serve as a general ground for quality assessment, we require each region to serve a defined purpose and have an unambiguous construction rule. We particularly oppose a naive definition of quality regions as the powerset of revisions. We evaluate the feasibility of our formal framework by implementing it as a software library and realizing it as a command-line tool for general use. We show how our tool works using an example use case.

D. Contribution

This work contributes a formal framework of quality regions in revision management systems. Therefore, we provide formal, graph-based definitions for each region type. We base our formalism on the well-known structure of version graphs. To facilitate the formulation and understanding of our definitions,

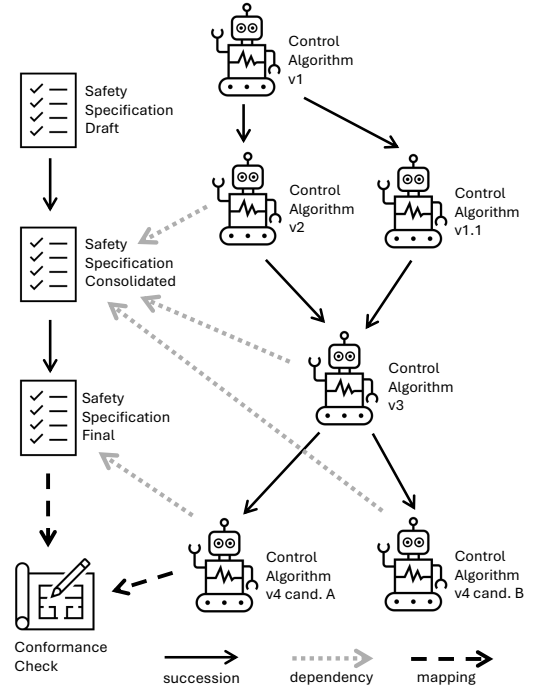


Fig. 1. Example of a system comprising two parts with linked revision histories. Both artifacts evolve independently but also depend on each other.

we formally introduce the concepts of revision graphs, revision graphs with merge edges, and multi-revision graphs in a step-by-step manner. Lastly, we contribute the software library *coconlib* and the command-line tool *cocon-cli* that implement and demonstrate our formal framework. Our software library is designed to extend existing revision management systems with quality region support. This work does not contribute a novel revision- (or version-) management system. On the contrary, this work aims to improve existing approaches for revision management by its developed concepts and tools.

E. Example

This section provides an illustrative example of an RMS with multiple histories. Figure 1 shows a simple system comprising a robotic control algorithm and a safety specification. The control algorithm may be realized as a state machine or an automaton. The safety specification may be a tabular document describing and rating different safety properties as a checklist. Both artifacts have revision histories. The safety specification has three revisions: the draft, the consolidation, and the final specification. The control algorithm is developed in a more agile manner. We see different intermediate revisions and a merge along the way to the algorithm’s fourth version. The control algorithm is related via dependencies on the safety specification. The dependencies are depicted as dotted arrows. Different revisions of the algorithm depend on different revisions of the safety specification. Lastly, we have the conformance check in the bottom left corner. This document is a temporary artifact for reviewing the conformance of the algorithms and the safety specification. It is derived from two

revisions. Resulting modifications might be propagated back to the original artifacts after the review.

II. BACKGROUND

This section provides a comprehensive background of this work's central topics: versioning, version control systems, and software quality assurance.

Versioning of software objects has been researched for several decades. Works from the database- and software product line communities described the terms *version*, *variant*, and *revision* [1], [5], [6]. These works already discovered the natural graph structure of a versioning system, which can be denoted in different representations, such as one- and two-level version graphs, version grids, or version matrices. The base relation between versions is the “successor” relation. Other relations, such as constraints or merges, may also exist [1].

We find deviating usages of the terms *version*, *variant*, and *revision* within the large corpus of existing literature. Software evolution and variation is commonly distinct in space and time dimensions [8]. Space refers to the dimension of variability, i.e., features or alternative developments. Time refers to the dimension of evolution, i.e., updating or revising existing software objects. This two-dimensionality implies that a software object is a version of another in time or a variant of another in space. However, some works consider the *version* to be the top-level element that can either be a *variant* or a *revision* [14]. Some more recent works aim to again unify different variability notions as *revisions* [7]. We decided to also refer to all the elements in a version graph as *revisions*, thus making it a *revision graph*. Depending on the observer's standpoint, a revision can then play the role of a *version* or *variant*, i.e., we consider them non-rigid types [15].

Different version control systems exist - or have existed - in practice [2]. The state-of-the-art software versioning tool is Git, enabling many different development styles and workflows [16]. Although Git works well with code, it has drawbacks when working in specialized domains such as software modeling or the versioning of complex systems comprising individually evolving parts, e.g., Cyber-Physical Systems. Therefore, different specialized RMS exist in research and practice. Schwägerl et al. [3] presented the tool *SuperMod* as an RMS for software product line engineering. The *Vitruvius* approach [4] addresses the problem of assuring consistency in multiple semantically overlapping development artifacts. Although the current version of *Vitruvius* manages interrelated revisions in the space dimension, it supports no versioning in time.

While the aforementioned systems and concepts enable the versioning and collaborative development of software, they do not guarantee the quality of the developed system. On the contrary, having a system with multiple revisions simultaneously in production makes quality assurance difficult. *Technical debt* builds up if quality properties are neglected [12], [13]. Different metrics are employed to measure and track the technical debt of a development project. A variety of quality properties and metrics exist [17]–[19]. In most cases,

they operate on a single revision of a system. However, from an organizational perspective, software evolution must be well-managed and controlled [20]. Therefore, one must be able to assess the quality of an RMS. While conducting previous work on a consistency metric for sets of variants [21], [22], we discovered that only few metrics and analysis techniques consider non-linear revision histories [23]–[25]. We expect this field to move into the focus of future research.

III. TERMINOLOGY

To clarify the following sections, this section introduces the necessary terminology. A version management system, or revision management system (RMS), stores the evolution history of a digital artifact. This artifact is typically a software system under development. We use the term *revision* to refer to the manifestations of the managed artifact at fixed points within its evolution history. We write “manifestation”, as the management system may work on a variety of principles. It could store actual snapshots, partial snapshots, or deltas, e.g., change operations. However, the internal working of the RMS is of no concern for this work.

A *revision* can be a version, a variant, or both at the same time, depending on the standpoint and objective of the observer. If the observer looks at an artifact's prior development history, it plays the role of a version. If the observer investigates alternative implementations of an artifact, it plays the role of a variant. A revision is identified by a unique name and directed acyclic *successor* relation to other revisions. Depending on the restrictions of the successor relation, this leads to the natural mathematical structure of a *revision tree* or *revision graph*. This concept is also commonly known as a *version graph* [1]. As we allow an RMS to manage multiple related artifacts, it may comprise several revision graphs. A second type of relation can link revisions of different revision graphs. We refer to this structure as a *multi-revision graph* which we further specify in Sec. V-D.

IV. REVISION TREES AND -GRAPHS

The literature defines different variations of revision graphs and trees [1]. This section presents this work's definition of revision graphs. However, we do not “reinvent” the idea of a revision graph but formally define it in a suitable way for the subsequent introduction of quality regions. The underlying structure of a revision graph is a *directed acyclic graph* (DAG). However, as described below, we restrict the DAG so that individual traversal functions operate on trees. Therefore, we first present the structure of a revision tree and extend it to a revision graph afterward. We further extend the revision graph to a multi-revision graph to fully support our definition of an RMS.

A. Revision Trees

We consider a set \mathcal{R} of all possible revisions and a subset $R \subseteq \mathcal{R}$ of managed revisions. Possible revisions are theoretically all revisions that could be created in the future or have been created in the past. Managed revisions are those

revisions that are actually stored in the RMS. The set R contains a distinguished element r_0 called the initial revision. Apart from the initial revision, each revision $r \in R$ admits a unique predecessor describing from which other revisions it was evolved. With the convention that the initial revision is its own predecessor, the predecessor relation can be described as a function $pred : R \rightarrow R$, with $pred(r_0) = r_0$. For r_0 to be indeed be the initial revision, we require that iteratively applying $pred$ always leads to r_0 . In other words, $pred$ induces a partial order on R with r_0 as the lower bound. We can view R as a directed rooted tree with the initial revision as the root. This is called a *revision tree*.

To navigate the revision tree, we define additional functions. The successor function $succ : R \rightarrow \mathcal{P}(R)$ maps each revision to the set of all its direct successors. It is essentially the preimage of the predecessor function. We write $pred^{(n)}$ for the n -th iteration of $pred$, such that $pred^{(n)}(r)$ is the n -th predecessor of r . For $n \in \mathbb{N}$, we also write $pred^{\leq n}(r)$ for the set of all predecessors of r up to n , i.e., $pred^{\leq n}(r) = \{pred^{(k)}(r) \mid k \leq n\}$ and $pred^*(r)$ for the set of all predecessors of r , i.e., $pred^*(r) = \{pred^{(n)}(r) \mid n \in \mathbb{N}\}$. We consider a function $root$ mapping any revision r to the initial revision, i.e., the root of the tree r_0 . Additionally, we define the set of all leafs of a revision r as $leafs(r)$. This set contains all revisions that can be reached from r via $succ$ and have no further successor. Formally, a leaf is a revision l such that $succ(l) = \emptyset$, and $leafs(r) = \{l \in R \mid r \in pred^*(l) \wedge succ(l) = \emptyset\}$.

In summary, we defined a revision tree as a (directed) rooted tree based on the $pred$ relation, together with the functions $pred^*$, $succ$, $leafs$, and $root$.

B. Revision Graphs

A revision tree supports the description of revisions with a single predecessor for each revision. This property does not allow a revision tree to model merges. A merge - or unification - occurs when a revision has two predecessors. In practice, a merge is created by combining changes from two revisions into a new revision. Formally, considering merges means replacing the predecessor function $pred : R \rightarrow R$ with $pred : R \rightarrow \mathcal{P}_{1,2}(R)$, where $\mathcal{P}_{1,2}(R)$ is the set of all subsets of R with cardinality 1 or 2, meaning that every revision has exactly one or two predecessors. Then, a revision tree becomes a rooted directed acyclic graph. We call this graph a *revision graph*. Figure 2 depicts an exemplary revision graph. For navigating the resulting merge structure, we introduce explicit *merge edges*. A merge edge connects the lowest common ancestor of the two predecessors of a merge revision with the merge revision itself. A merge edge is visualized as a double-lined arrow. For instance, (a, e) is a merge edge in Figure 2. Figure 3 depicts an evolution from g_2 to g'_2 introducing the merge revision d as successor of both b and c . The merge edge is added in between a and d as a is the lowest common ancestor of b and c . Figure 4 shows two slightly larger revision graphs with merge revisions and merge edges.

Analyzing a revision graph with merges, the $pred$ function becomes ambiguous. We resolve this issue by making merge

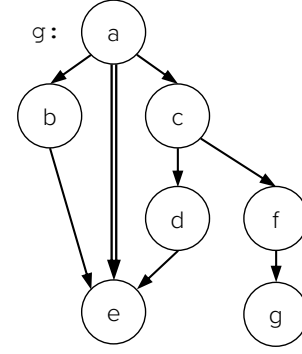


Fig. 2. Example of a revision graph with seven revisions and one merge edge.

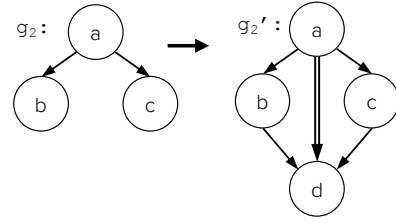


Fig. 3. Evolution scenario with a merge. The revisions b and c are merged into the new revision d . The merge is denoted via a merge-edge (double-line arrow).

edges “priority” edges. While traversing a merge node with ambiguous predecessors, the merge edge is traversed instead of the default edges. Consequently, traversing a revision graph becomes the same as traversing a revision tree. This technique is similar to highway hierarchies in path-finding algorithms [26], limited to two levels of priorities. In terms of a practical RMS, this means that the traversal functions considers merges as “squash merges,” while the actual history is retained. We raise the awareness that this mechanism “looses” the information about what happens on the distinct branches of a merge. To counter this loss, a separate function including all these revisions may be required. We leave such a definition open at this point.

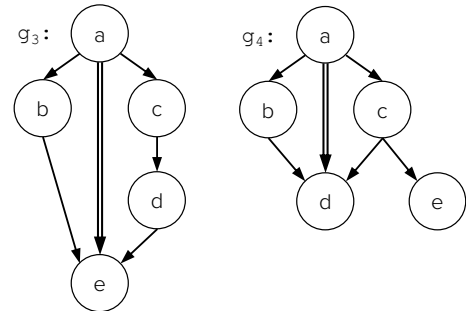


Fig. 4. Two examples of more complex merge scenarios. Merge partners are fully qualified revisions that can further evolve in space and time.

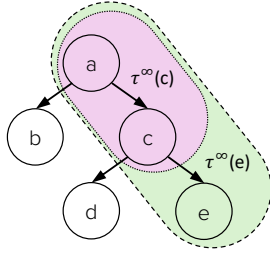


Fig. 5. Example of a revision graph with two marked time regions: The unbounded time region of e , $\tau^\infty(e)$, shown in green, and the unbounded time region of c , $\tau^\infty(c)$, shown in violet.

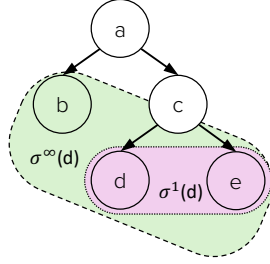


Fig. 6. Example of a revision graph with two marked space regions: The 1-bounded space region of d , $\sigma^1(d)$, shown in violet, and the unbounded space region of d , $\sigma^\infty(d)$, shown in green.

V. QUALITY REGIONS

This section introduces *quality regions* based on the previously introduced formalisms of revision graphs for the dimensions of time and space, for relations across multiple revision graphs, and for building projections/views.

A. Time Regions

A revision can be analyzed with its predecessors, i.e., previous versions, to reason about a quality property in time. Quality analysis over time is a common task in software engineering. Examples are the detection of breaking changes, the analysis of change patterns, tracking technical debt [12], [13], or the detection of coupled evolution [27]. All these analysis operate on a (possibly bounded) list of revisions sliced out of the revision graph.

We call such a slice of a revision graph a *time region* $\tau \subseteq R$ and distinguish *i*-bounded- τ^i and *unbounded time regions* τ^∞ . We define the unbounded time region of a revision r in a revision graph g as the set $\text{pred}^*(r)$. We define the bounded time region bound by $i \in \mathbb{N}$ as the set $\text{pred}^{\leq i}(r)$. In other words, the unbounded time region is the set of all predecessors of r , and the bounded time region is the set of the newest i predecessors of r . Looking at Figure 5, we write $\tau^\infty(e) = \{a, c, e\}$ and $\tau^\infty(c) = \{c, a\}$. The 1-bounded time region of e would be $\tau^1(e) = \{c, e\}$, applying pred one time.

B. Space Regions

A revision can be analyzed together with its n -closest neighbors, i.e., variations in space. The interpretation of this region depends on the semantics of the variants. They could

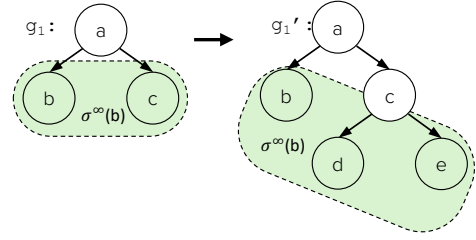


Fig. 7. Evolution scenario of a revision graph. The two revisions d and e are added. The unbounded space region of b evolves as well.

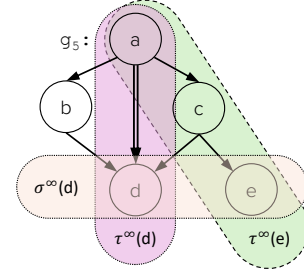


Fig. 8. Time- and space regions in a revision graph with a merge edge. The merge edge only influences the construction of $\tau^\infty(d)$.

be modified clones, features, alternatives, or temporary copies under change. As this work's revision graphs describe evolution histories rather than feature models, it becomes evident that our space dimension is rooted in the time dimension. In other words, the space is spanned by parallelism, or non-strict ordering, in time. Prominent analysis cases in space are the detection of inconsistencies, e.g., merge conflicts, incompatibilities, constraint violations or *drift* [22]. These analysis operate on a (possibly bounded) set of revisions co-existing in space. This can, for example, be the set of all branch heads in a Git repository which realize bug fixes.

On an abstract level, we call such a region a *space region* $\sigma \subseteq R$ and distinguish *i*-bounded- σ^i and *unbounded space regions* σ^∞ . We define the unbounded space region as the set of all leafs of the revision's *root*. Consequently, this region is equal for all revisions within the graph. We define the space region bounded by $i \in \mathbb{N}$ as the set of leafs given by the *leafs* function for the i 'th predecessor.

Figure 6 illustrates this definition by showing two space regions: $\sigma^\infty(d) = \{d, e, b\}$, and $\sigma^1(d) = \{d, e\}$. According to the definition, we can also write $\sigma^\infty(d) = \sigma^2(d)$. Figure 7 shows the evolution of the revision graph g_1 into the revision graph g_1' . Two new revisions were added as successors of c . The space region $\sigma^\infty(b)$, i.e., $\{b, c\}$, evolves to be $\{b, d, e\}$.

C. Regions Construction with Merge Edges

Constructing regions in revision graphs with merge edges differs in only one aspect from the construction in revision graphs without merge edges. If the traversal queries the predecessor of a revision that is a merge revision, the merge edge is prioritized. Thus, the path to the root can always be resolved without ambiguity. Figure 8 shows the resolution of regions on

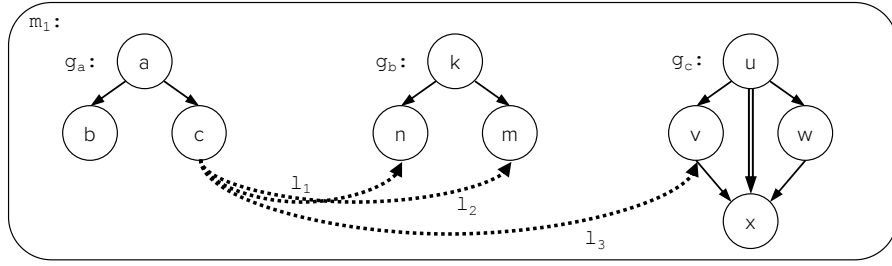


Fig. 9. Example of a multi-revision graph. The multi-revision system m comprises three revision graphs $g_1 \dots g_3$. The revision graphs must not overlap. Each graph has its individual root. Revisions may have links between each other. Three exemplary links are denoted as l_1, l_2, l_3 .

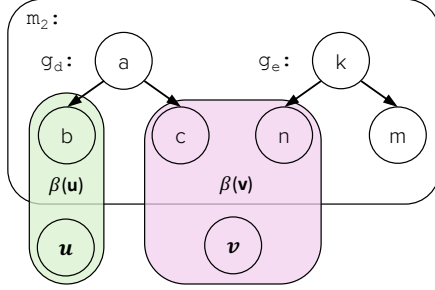


Fig. 10. Example of a multi-revision system comprising two revision graphs with dirty revisions u and v . They may become part of future revisions or may be discarded.

a revision graph with a merge edge. In this example, we show the unbounded time regions of d and e : $\tau^\infty(d) = \{d, a\}$ and $\tau^\infty(e) = \{e, c, a\}$. We also show the unbounded space region of d : $\sigma^\infty(d) = \{d, e\}$. The set $\text{pred}^*(e)$ of all predecessors of e is not influenced by the merge edge. The set $\text{pred}^*(d)$ of all predecessors of d exploits the prioritized merge edge (a, d) , such that $\text{pred}(d) = a$.

D. Multi-Revision Graphs

We allow a revision management system to handle the - possibly related - revision histories of more than one artifact. Examples of such systems are software projects containing application parts in different Git repositories or software models managed in a multi-model [4], [28], [29]. Such systems form sets of revision graphs with superimposed links, as illustrated in our introductory example. A link between revisions of different graphs is a directed edge with an arbitrary semantic, e.g., constraints, dependencies, or import relationships. A link must not have the semantics of a predecessor or successor, as used within a revision graph. We call the tuple of a set of revision graphs and a set of links a *multi-revision graph*. This surrounding structure does not influence the time- and space regions within the comprised revision graphs. Figure 9 shows a multi-revision graph $m_1 := (\{g_a, g_b, g_c\}, \{l_1, l_2, l_3\})$.

E. Volatile Regions

When editing an artifact managed by an RMS, a developer chooses a revision they want to evolve and creates a local version of it. This local copy is then edited until the developer

adds the changed artifact as a new revision to the RMS. It might also be possible that a developer discards their changes. We call the edited artifact “dirty” or “volatile” as long as it remains a working copy which is not yet checked into the RMS. However, the RMS may already know of its existence. It is also possible that a client jointly edits (a projection of) multiple artifacts in a volatile working copy. In software modeling, this is a subtype of *view-based* development [30], [31].

Within our framework of quality regions, we also aim to cover the analysis case of potentially long-living volatile artifacts. These artifacts may be views comprising different revisions as sources. Making them part of a quality region is important for assessing quality attributes a priori. This way, it becomes possible to describe the process of pre-calculating or “guessing” violations of a quality property before they manifest in the revision graph. An example is the preemptive detection of possible merge conflicts with other revisions [32].

We define the set of volatile revisions as a subset of the set of managed revisions $V \subseteq R$ it is derived from, i.e., has ties with. A volatile revision can but does not have to be an element of \mathcal{R} . However, the possible set of volatile revisions must not be smaller than \mathcal{R} itself. This allows it to live outside the revision management system. We define the *volatile region* β as a tuple comprising a volatile revision and the set of its sources $\beta := V \times \mathcal{P}(R)$. Figure 10 depicts a multi-revision system m_2 comprising two revision graphs g_d and g_e . There exist two volatile revisions u and v . They span the volatile regions $\beta(u) = (u, \{b\})$ and $\beta(v) = (v, \{c, n\})$.

F. Overview

Figure 11 summarizes the introduced regions in a single diagram. The figure shows a multi-revision graph comprising two revision graphs A and B . A single link exists between the revisions q of A and d of B . The right side of the figure highlights time and space regions for the revision k . The figure also highlights the volatile region of the temporary revision/view v . Of course, the highlighted regions are only examples based on specific revisions and may be constructed for any revision in the multi-revision graph.

Lastly, we point out that the introduced regions of space and time, as well volatile regions are (for non-trivial revision graphs) a subset of $\mathcal{P}(R)$, meaning that additional regions can

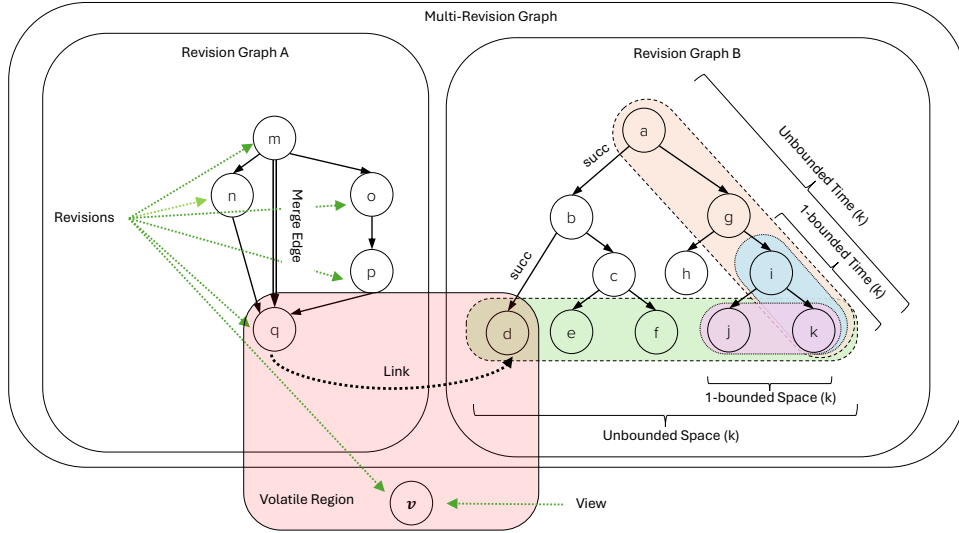


Fig. 11. Overview of (multi-revision) graphs and the introduced quality regions. The shown regions in time and space are constructed for the revision k .

be defined and constructed. The existence of such regions is not a problem for our framework. On the contrary, we motivate and support the definition of additional meaningful quality regions based on our formal framework.

VI. PROOF OF CONCEPT

This work's main contribution is the formal definition of quality regions for revision management systems. However, we also want to show that this theory can be applied in practice. Specifically, we aim to demonstrate how the proposed theoretical framework can be realized as a programming library and ready-to-use tool. The implementation serves as a proof of concept for the theoretical concepts introduced in Sections IV and V. For this purpose, we underwent the following steps: First, we derived a set of usecases for interacting with revision graphs and quality regions; Second, we decided on the technical space to implement the proof of concept; Third, we implemented a library called *coconlib* that supports the usecases; Fourth, we implemented a command line tool called *cocon-cli* that allows users to interact with the library and create persistent revision graphs and quality regions. Finally, we realized the running example from Section I-E as a small case study for the library and the command-line tool. All tools and examples developed in this work are publicly available (see Sec. VII).

A. Separation from Existing Works

This section describes the implementation of multi-revision graphs and quality regions in the form of a programming library. This work does not aim to develop a novel revision management system. The provided library is not intended to replace any existing functionality of existing revision management systems. The purpose of the developed library and tools is to provide a programming interface for creating and manipulating multi-revision graphs and quality regions, facilitating experimentation and research. Furthermore, we are

confident that the provided library can be used to extend existing revision management systems with quality region support. For example, we envision the developed tooling to be used on top of applications such as Git for extending their features and enabling the uniform querying of the underlying revision graph.

B. Goal

A (multi-)revision system manages a development project's history. A (multi-)revision graph abstracts the revisions and their relationships inside the RMS. The RMS may explicitly maintain the graph, or it can be constructed on demand by analyzing an existing revision structure. In both cases, the revision graph has to be updated incrementally. An implementation must, therefore, support CRUD operations on the revision graph, i.e., its nodes and edges. As a multi-revision system contains multiple revision graphs, creating, updating, and deleting them must be possible. A set of default queries must be provided to observe the structure of the revision graphs, e.g., list all revisions, relations, etc. Lastly, it must be possible to query the quality regions. The system's implementation should be extensible and make use of existing libraries. The created revision graphs must be persistable in a serialized format.

C. Realization

The following sections describe the implementation of the developed library and the CLI tool. We refer to the source code, library, and tool documentation for a more detailed description in Sec. VII. Furthermore, all developed tools are available open-source¹. Both the library and the CLI tool are implemented in Kotlin. Consequently, both run on the Java Virtual Machine (JVM). The library can be used with any JVM-based language, e.g., Java, Scala, or Kotlin.

¹<https://github.com/KKegel/coconlib>

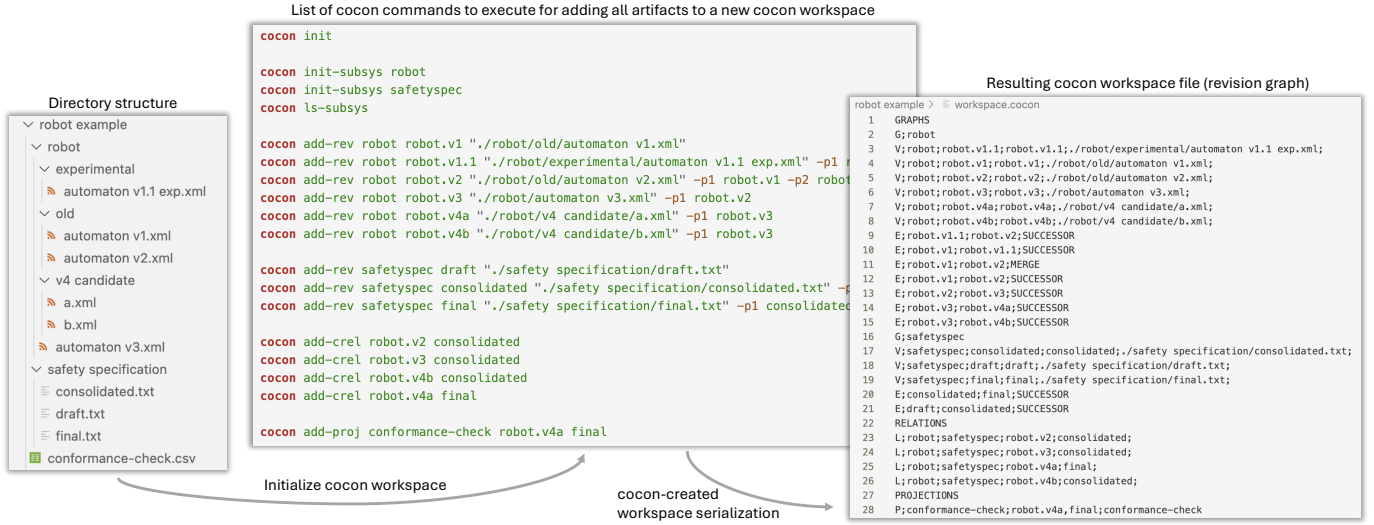


Fig. 12. Overview of an exemplary workspace setup process using *cocon-cli*. The part depicts the workspace’s file structure. The actual content of the files is of no importance. The middle part shows the executed setup commands. The right side shows the inside of the created workspace file, i.e., the multi-revision graph with all its vertices and edges.

D. coconlib

We implemented the *coconlib* programming library for working with revision graphs and quality regions. The *coconlib* provides programming interfaces to create and manipulate revision graphs, query a revision graph’s contents, and query quality regions. We used *Apache Tinkergraph* as the framework for realizing the underlying graph structure. Tinkergraph is a lightweight graph engine that supports graphs with properties. The Tinkergraph framework comes with the *Gremlin* query language [33], which allows querying, traversing, and manipulating the graph. The *coconlib*’s API is designed to be extensible and provides a stage-wise level of technical abstraction. The highest level comprises revision-graph operations with error handling and validation. On this API level, it is possible to construct, manage, and query revision graphs. As this level is implemented “safe”, the library assures that invalid operations are handled properly without invalidating the graph structure. Suppose the client aims to implement custom graph traversals or queries, such as adding additional custom quality regions or extending the revision graph with extra properties or constraints. In that case, they can access the underlying graph structure directly and manipulate it using the Gremlin language. However, in this case, it lies with the developer to ensure the validity of the graph structure.

E. cocon-cli

For experimenting with the *coconlib* beyond basic testing, we implemented the *cocon-cli*. The *cocon-cli* tool is a command-line interface to the *coconlib* library. *cocon-cli* uses a workspace-based approach, i.e., it operates on a local file structure. The created workspace serves as a wrapper or placeholder for any arbitrary revision management system. In the default case, the wrapped RMS is just the user’s local file system. In other words, the *cocon-cli* does not actually manage

revisions, but only manages a revision graph consisting of pointers to the actual revisions. The user can create a new *cocon* workspace by executing the *init* command locally. The workspace’s revision graphs can be populated, modified, and queried using a set of commands. Executing a command will load the workspace descriptor, perform the requested operation, and save the workspace descriptor again. The tool will abort in case an invalid command or graph invalidation is detected. We show exemplary commands in the following section..

F. Example Workflow

We use our illustrative example from Section I-E to demonstrate the functionality of our proof of concept. For the sake of simplicity, we use a single “dummy” file per artifact, i.e., revision. This is a valid simplification, as our tooling works solely as a wrapper and does not process the managed files. However, each managed revision/artifact can be anything that can be represented as a path, including whole directories or repositories. The set-up file structure is shown in Figure 12 on the left side. We assume that the revisions are not managed by any other VCS such as Git. Interacting with revisions stored within Git repositories (commits) is an interesting use case which we consider for future work, as this would require a quite complex technical integration which goes beyond the scope of this work’s proof of concept. We then executed a sequence of CLI commands to create a new workspace and add the revisions and their relationships. As the workspace is persistent, this has to be done only once. The command *cocon* is our alias for the locally installed *cocon-cli* tool. The executed commands are shown in Figure 12 in the middle of the figure. We aggregated the commands into a single bash script for the sake of simplicity. Executing the script or commands on their own on a command-line makes no difference. Each

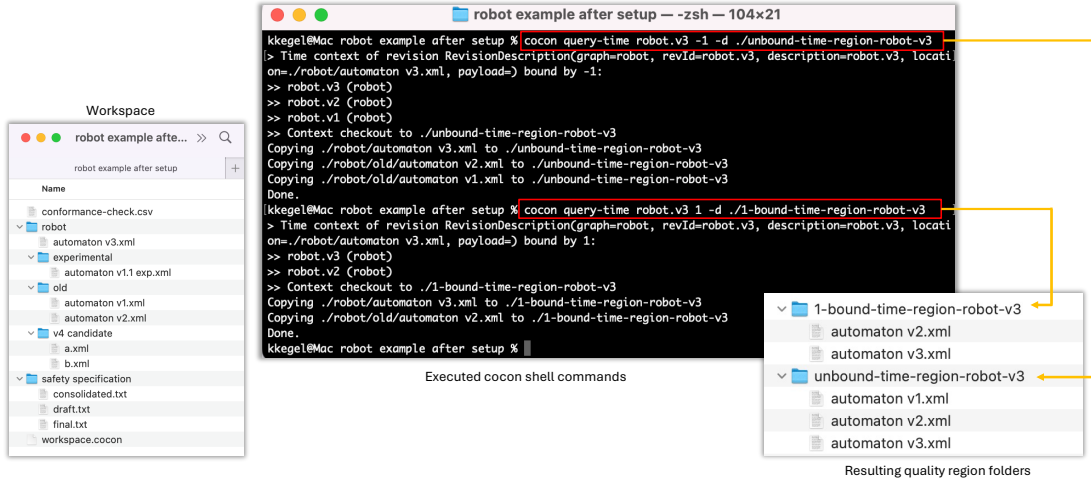


Fig. 13. Exemplary region query and checkout process using the workspace initialized in Fig. 12. The left side shows the local file structure with the workspace file. The middle window shows a terminal with two executed region queries (bounded- and unbounded time). The right side shows the directories created by the region queries containing all participating revisions.

command opens the workspace, performs its action, and closes the workspace afterwards. We see commands for initializing the two revision graphs (*init-subsys*), adding the revisions (*add-rev*) and their relationships (*add-crel*). The succession structures and merge structures are automatically built from the given predecessors. The right side of Figure 12 shows the resulting serialized multi-revision graph in the workspace file. Figure 13 shows an example of two region queries with checkout. The library and tool support querying all quality regions introduced in this work. The example retrieves the unbound time region (denoted -1) and the 1-bounded time region for the revision *robot.v3*. The region participants' ids are printed. Because the -d option is set, the tool performs a checkout of all revisions to a specified directory. As our tooling operates on top of the file system, a "checkout" is simply realized as collecting the target revisions by their file paths and copying them to the region's target location. The resulting directory is shown in the right side of Figure 13. The result can now be used for quality analysis within the region, i.e., passing this directory as input to an analysis tool.

VII. REPRODUCIBILITY

We provide a reproduction package as supplementary material [34] hosted on ZENODO. The package includes the used versions of the software tools, as well as the shown example case with all information required for reproduction. Furthermore, the *coconlib* and *cocon-cli* are publicly available on GitHub under an open-source license. We included references to the individual repositories within the supplementary material.

VIII. CONCLUSION

This work presented an abstract framework of quality regions based on a formal definition of revision graphs. Revision graphs, also known as version graphs, are a well-studied structure used to represent the development history of a software

object. They play a central role as a data structure in revision management systems. Based on the revision graph's formal structure, we proposed a framework of quality regions for reasoning about quality properties of a versioned system. These regions serve both as a nomenclature and as a conceptual basis for future research on quality assurance and metrics for versioned systems. The presented formal framework operates at an abstraction level suitable for any concrete versioning system, as long as the graph-based representation of the revision structure is applicable. However, we see the largest impact of our framework in the field of model management, as model management systems emphasize more advanced and fine-grained versioning mechanisms compared to code versioning with Git.

We showcased the technical feasibility of our formalism by implementing a framework able to represent, construct and query revision graphs and their quality regions. Our framework called *coconlib* implements all features of the proposed formalism and is available as an open-source library. Furthermore, we developed a lightweight command-line application *cocon-cli* based on the library. The implementation, particularly the *coconlib* library, is extensible and can be used in other tools and applications. We consider our library a ready-to-use tool for researchers and practitioners. In summary, we answered this work's research question by the given region definitions and constructions in Section V and the subsequent proof of concept in Section VI.

Future work will focus on both extending the theoretical foundation of our framework and extending the implementation. On the theoretical side, we will extend the formal foundations of this work to capture properties of revision graphs in greater detail. In particular, we aim to express more refined revision relations, which may be constrained or attributed. On the practical side, we want to analyze how different metrics perform in different regions. With the theo-

retical foundations laid out in this work, it becomes possible to analyze the quality properties of a versioned system clearly and systematically. For example, we aim to investigate the distribution of inconsistencies in a revision graph using the *drift* metric [22] across different quality regions. Furthermore, we aim to analyze and benchmark different quality metrics in model- and code repositories across different regions in space and time to investigate the importance of revision-graph aware quality assurance.

ACKNOWLEDGMENT

This work is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1608 – 501798263.

REFERENCES

- [1] R. Conradi and B. Westfechtel, “Version models for software configuration management,” *ACM Computing Surveys (CSUR)*, vol. 30, no. 2, pp. 232–282, 1998.
- [2] L. Linsbauer, F. Schwägerl, T. Berger, and P. Grünbacher, “Concepts of variation control systems,” *Journal of Systems and Software*, vol. 171, p. 110796, 2021.
- [3] F. Schwägerl and B. Westfechtel, “Supermod: tool support for collaborative filtered model-driven software product line engineering,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 822–827.
- [4] H. Klare, M. E. Kramer, M. Langhammer, D. Werle, E. Burger, and R. Reussner, “Enabling consistency in view-based system development — the vitruvius approach,” *Journal of Systems and Software*, vol. 171, p. 110815, 2021.
- [5] K. R. Dittrich and R. A. Lorie, “Version support for engineering database systems,” *IEEE Transactions on Software Engineering*, vol. 14, no. 4, pp. 429–437, 1988.
- [6] E. Sciore, “Versioning and configuration management in an object-oriented data model,” *The VLDB journal*, vol. 3, pp. 77–106, 1994.
- [7] S. Ananieva, S. Greiner, T. Kühn, J. Krüger, L. Linsbauer, S. Grüner, T. Kehrer, H. Klare, A. Koziolok, H. Lönn, S. Krieter, C. Seidl, S. Ramesh, R. Reussner, and B. Westfechtel, “A conceptual model for unifying variability in space and time,” in *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*, ser. SPLC ’20. New York, NY, USA: Association for Computing Machinery, 2020.
- [8] K. Pohl, G. Böckle, and F. Van Der Linden, *Software product line engineering: foundations, principles, and techniques*. Springer, 2005, vol. 1.
- [9] C. Seidl, I. Schaefer, and U. Aßmann, “Integrated management of variability in space and time in software families,” in *Proceedings of the 18th International Software Product Line Conference - Volume 1*, ser. SPLC ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 22–31.
- [10] *ISO/IEC 25010 Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - Product quality model*, Std., 2023-11-00.
- [11] *ISO/IEC 25023 Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - Measurement of system and software product quality*, Std., 2016-06-00.
- [12] C. Seaman and Y. Guo, “Measuring and monitoring technical debt,” in *Advances in Computers*. Elsevier, 2011, vol. 82, pp. 25–46.
- [13] P. Kruchten, R. L. Nord, and I. Ozkaya, “Technical debt: From metaphor to theory and practice,” *IEEE Software*, vol. 29, no. 6, pp. 18–21, 2012.
- [14] B. Westfechtel, B. P. Munch, and R. Conradi, “A layered architecture for uniform version management,” *IEEE Transactions on Software Engineering*, vol. 27, no. 12, pp. 1111–1133, 2001.
- [15] G. Guizzardi, “Ontological foundations for structural conceptual models,” 2005.
- [16] J. C. Cortés Ríos, S. M. Embury, and S. Eraslan, “A unifying framework for the systematic analysis of git workflows,” *Information and Software Technology*, vol. 145, p. 106811, 2022.
- [17] H. Nakai, N. Tsuda, K. Honda, H. Washizaki, and Y. Fukazawa, “Initial framework for software quality evaluation based on iso/iec 25022 and iso/iec 25023,” in *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2016, pp. 410–411.
- [18] T. Mens and S. Demeyer, “Future trends in software evolution metrics,” in *Proceedings of the 4th International Workshop on Principles of Software Evolution*, ser. IWSE ’01. New York, NY, USA: Association for Computing Machinery, 2001, p. 83–86.
- [19] M. Staron, “Metrics for Software Design and Architectures,” in *Automotive Software Architectures*. Cham: Springer International Publishing, 2021, pp. 215–233.
- [20] F. J. Furrer, *Future-proof software-systems*, ser. Springer eBooks. Springer Vieweg, 2019.
- [21] K. Kegel, S. Götz, R. Marx, and U. Aßmann, “A variance-based drift metric for inconsistency estimation in model variant sets,” vol. 23, no. 3, Jul. 2024, pp. 1–14, the 20th European Conference on Modelling Foundations and Applications (ECMFA 2024).
- [22] K. Kegel, S. Götz, and U. Aßmann, “Branch drift: A visually explainable metric for consistency monitoring in collaborative software development,” *IEEE Access*, pp. 1–1, 2025.
- [23] D. Rozenberg, I. Beschastnikh, F. Kosmale, V. Poser, H. Becker, M. Palyart, and G. C. Murphy, “Comparing repositories visually with repograms,” in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 109–120.
- [24] E. Shihab, C. Bird, and T. Zimmermann, “The effect of branching strategies on software quality,” in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM ’12. New York, NY, USA: Association for Computing Machinery, 2012, p. 301–310.
- [25] V. Kovalenko, F. Palomba, and A. Bacchelli, “Mining file histories: should we consider branches?” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 202–213.
- [26] P. Sanders and D. Schultes, “Highway hierarchies hasten exact shortest path queries,” in *Algorithms – ESA 2005*, G. S. Brodal and S. Leonardi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 568–579.
- [27] H. Gall, K. Hajek, and M. Jazayeri, “Detection of logical coupling based on product release history,” in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, 1998, pp. 190–198.
- [28] P. Stünkel, H. König, Y. Lamo, and A. Rutle, “Multimodel correspondence through inter-model constraints,” in *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming*, ser. Programming ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 9–17.
- [29] Z. Diskin, Y. Xiong, and K. Czarnecki, “Specifying overlaps of heterogeneous models for global consistency checking,” in *Proceedings of the First International Workshop on Model-Driven Interoperability*, ser. MDI ’10. New York, NY, USA: Association for Computing Machinery, 2010, p. 42–51.
- [30] H. Bruneliere, E. Burger, J. Cabot, and M. Wimmer, “A feature-based survey of model view approaches,” *Software & Systems Modeling*, vol. 18, no. 3, pp. 1931–1952, 2019.
- [31] A. Cicchetti, F. Ciccozzi, and A. Pierantonio, “Multi-view approaches for software and system modelling: a systematic literature review,” *Software and Systems Modeling*, vol. 18, no. 6, pp. 3207–3233, 2019.
- [32] K. Kegel, A. Domanowski, K. Feichtinger, R. Pascual, and U. Aßmann, “A delta-oracle for fast model merge conflict estimation using sketch-based critical pair analysis,” in *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS Companion ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1046–1055.
- [33] M. A. Rodriguez, “The Gremlin graph traversal machine and language (invited talk),” in *Proceedings of the 15th Symposium on Database Programming Languages*, ser. DBPL 2015. New York, NY, USA: Association for Computing Machinery, Oct. 2015, pp. 1–10.
- [34] K. Kegel, “Supplementary material / implementation: Coconlib and cocon-cli,” Jul. 2025. [Online]. Available: <https://doi.org/10.5281/zenodo.15854631>