Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

Visualisierung von veränderten grafischen Modellen und Diagrammen im Rahmen der Überprüfung von Modellen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Florian Zoubek, BSc
Matrikelnummer 0828559

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: O.Univ.Prof. Dipl.-Ing. Mag. Dr.techn. Gerti Kappel
Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.rer.soc.oec. Tanja Mayerhofer, BSc
Dipl.-Ing. Dr.techn. Philip Langer

Wien, 7. April 2018

Florian Zoubek	Gerti Kappel

Visualization of Evolving Graphical Models and Diagrams in the Context of Model Review

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Florian Zoubek, BSc
Registration Number 0828559

to the Faculty of Informatics

at the TU Wien

Advisor: O.Univ.Prof. Dipl.-Ing. Mag. Dr.techn. Gerti Kappel
Assistance: Univ.Ass. Dipl.-Ing. Dr.rer.soc.oec. Tanja Mayerhofer, BSc
Dipl.-Ing. Dr.techn. Philip Langer

Vienna, 7th April, 2018

Florian Zoubek | Gerti Kappel

Erklärung zur Verfassung der Arbeit

Florian Zoubek, BSc
Viktor Kaplanstraße 13, 3013 Tullnerbach

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 7. April 2018

Florian Zoubek

Danksagung

An dieser Stelle möchte ich mich bei allen Bedanken, die mich während des Verfassens der Arbeit mit Rat und Motivation sowie bei der Evaluierung unterstützt haben. Allen voran möchte mich bei meinen Eltern für die Ermöglichung des Studiums, Rat, Motivation und nicht zu guter Letzt für die finanzielle Unterstützung während meines Studiums bedanken. Auch bei meinem Bruder möchte ich mich ausgiebig bedanken, der während dieser Zeit immer hinter mir stand. Diese Zeit war durch positive, aber auch einige negative Ereignisse geprägt, weshalb ich mich generell bei meinen Eltern und meinem Bruder auch für die Unterstützung und den Rückhalt während dieser Zeit bedanken möchte.

Mein Dank gilt auch den Betreuerinnen dieser Arbeit an der Technischen Universität Wien. Ich möchte mich auch bei Philip Langer für die Mitbetreuung dieser Arbeit bedanken, der immer bemüht war meine Fragen zu beantworten. Des Weiteren möchte ich mich bei meinen Studienkollegen, ehemaligen Schulkollegen und den weiteren Teilnehmern meiner Fallstudie für die investierte Zeit bedanken.

Gewidmet meiner kürzlich verstorbenen Großmutter Erna Fiedler. Sie half wo sie nur konnte und schaffte es immer wieder mich zu motivieren alle Hindernisse zu überwinden, so unüberwindlich sie auch schienen.

Acknowledgements

I would like to thank all persons at this point who had supported me during the evaluation and during the process of writing this thesis with advice and feedback. First of all I would like to thank my parents for the possibility to study, as well as for advices, motivation and also for the financial support during my study. Also I would like to thank my brother, who also backed me during this time. Positive, but also negative events occurred during the last years and so again I would like to thank my parents and my brother also for the support in this timespan.

Also many thanks to all advisors of this thesis at the university of technology in Vienna. I would also like to thank Philip Langer for the feedback about my work and who did not hesitate to answer my questions. Last but not least I would like to thank all of my study colleagues, former classmates and all other participants of the case study for the invested time.

Dedicated to my grandmother Erna Fiedler who recently passed away. She helped wherever she could, and always motivated me to break through all obstacles, regardless how unconquerable they seemed.

Kurzfassung

Code Reviews werden in vielen aktuellen Softwareprojekten zur Prüfung von Änderungen eingesetzt. Ziel dieser Code Reviews ist es Fehler, die durch Änderungen an dem Code entstanden sind, frühzeitig zu erkennen und zu verhindern, bevor diese in das Projekt integriert werden. Zur Unterstützung dieses Überprüfungsprozesses existieren bereits verschiedene Review Tools. Diese erlauben es einzelne, problematische Teile des Quelltextes zu markieren, kommentieren und mit anderen Personen zu diskutieren, sowie den Verlauf des Reviews nachzuvollziehen. Zusätzlich erlauben sie es auch die Änderungen zurückzuweisen oder anzunehmen. Da, wie der Name schon andeutet, Quelltext auf Text basiert, sind diese Tools primär auf Text-Daten ausgerichtet. Die Artefakte die im Zuge eines Reviews betrachtet werden müssen, sind jedoch nicht nur auf Quelltext beschränkt, sondern betreffen neben anderen Artefakten oft auch Modelle und Diagramme. Das inkludiert auch Diagramme und Modelle die durch eine grafische Syntax beschrieben werden, jedoch nicht notwendigerweise eine für den Benutzer lesbare textuelle Repräsentation besitzen. Ein Beispiel dafür sind Papyrus Unified Modeling Language (UML) Diagramme. Daher werden die Visualisierungsmethoden existierender Review Tools für diese Art von Modellen als nicht zufriedenstellend eingestuft. Stattdessen werden diese Review Tools mit anderen Tools kombiniert, die diese Art von Modellen vergleichen können, aber den Überprüfungsprozess nicht unterstützen. Dies erhöht jedoch auch die Komplexität und bedeutet einen höheren Aufwand, der sich negativ auf den Überprüfungsprozess auswirken kann. In dieser Diplomarbeit werden Visualisierungen und Interaktionstechniken vorgestellt die Entwickler bei dem Überprüfungsprozess von Diagrammen und Modellen, die durch eine grafische Syntax beschrieben werden, unterstützen. Verwandte Techniken, sowie existierende Methoden zum Vergleich von Modellen und für Code Reviews bilden die Basis für die entwickelten Visualisierungen und Interaktionstechniken. Zur Untermauerung dieser Visualisierungen und Interaktionstechniken wird eine Fallstudie präsentiert, die mit einem Prototyp durchgeführt wurde der auf dem Eclipse Modeling Framework aufbaut.

Abstract

Code reviews are used today in several software projects to check changes before they are merged. The aim of code reviews is to identify and prevent mistakes introduced by changed artifacts before they are merged into the actual software project. Therefore, several tools exist to support developers with the reviewing process. Such tools usually allow to mark, comment and discuss parts of the code that are considered to be problematic, as well as track the history of the reviewed changes. Of course they also support approval and rejection of changes. As source code is usually text based, these tools focus on source code or text-based artifacts. However, like almost every artifact in a software development project, models also change over time. This also includes diagrams, or models with a graphical syntax, which may not necessarily have a textual representation that is understandable by the average user. Papyrus Unified Modeling Language (UML) Diagrams are an example for such models. Therefore the techniques and visualizations of existing tools are usually considered to be unsatisfactory for such models by developers. So developers have to combine the code review tool with comparison tools that support the given model types, but do not support the review process. However, this adds additional overhead and complexity that may affect the review negatively. This thesis provides visualizations and interaction techniques that support developers who need to handle diagrams as well as models with a graphical syntax during the review process. These visualizations and techniques are obtained by analyzing existing methods for model comparison, code review and related techniques that might support the review process. A case study is presented using a prototype implementation utilizing frameworks based on the Eclipse Modeling Framework to prove the value of the proposed visualizations and techniques.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**
1.1 Motivation 1
1.2 Aim of the Thesis 3
1.3 Methodological Approach 4
1.4 Structure of this Thesis 4

2 State of the Art **7**
2.1 Existing Code Review Tools 7
2.2 Review Process & Artifacts 21
2.3 Graphical Models and Diagrams 24
2.4 Existing Difference Visualization Techniques 25
2.5 Element Annotation Visualization Techniques 28
2.6 Review History Visualization Techniques 29

3 Mervin: A Graphical Model and Diagram Review Tool **31**
3.1 Difference Types 34
3.2 Unified Difference View 35
3.3 Property Differences View 48
3.4 Comment View 50
3.5 Version History View 52
3.6 Review Explorer View 55
3.7 View Coordination And Highlighting 59
3.8 Required Input Data 60

4 Implementation **63**
4.1 Papyrus and GMF Overview 64
4.2 EMF Compare Overview 66
4.3 Internal Review Model 68

4.4 Gerrit Integration 71
4.5 EMF Compare Difference Mapping 74
4.6 Obtaining Layout Information 77
4.7 Unified View Implementation 78
4.8 Difference History 87
4.9 Highlight Contexts 90

5 Evaluation **93**
5.1 Evaluation Setup 96
5.2 Evaluation Results 98
5.3 Summary 106

6 Conclusion **107**
6.1 Future Work 107
6.2 Summary 109

7 Appendix **111**
7.1 Case Examples 111

List of Figures **115**

List of Tables **117**

List of Algorithms **119**

Index **121**

Glossary **127**

Acronyms **131**

Bibliography **133**

CHAPTER 1

Introduction

1.1 Motivation

Code reviews and tools designed to support code reviews [Goo17b, Pha17, Bea17, Rho17, Atl17b] are part of many large scale software development processes. Gerrit is an example for such a code review tool, which is used in popular open source projects such as the Android Open Source Project [Pro17], Eclipse [Ecl17a], LibreOffice [Fou17] and many others [Goo17a]. The aim of code reviews is to identify and prevent mistakes introduced by changed artifacts before they are merged into the actual software project. For this purpose, one or more other persons or automated processes act as reviewers. They examine the changed artifacts and provide some form of feedback, such as comments on changes or an overall rating of the proposed changes. How such reviews are done varies and ranges from pair programming to formal inspections. However, the basic review process supported by most of the review tools is as follows: A contributor proposes changes to the current set of artifacts, which aim to solve a particular task. After that, one or more reviewers provide some sort of feedback and decide whether the contribution will be accepted or not. If a contribution is not accepted, the contributor may provide new changes to the set of artifacts based on the given feedback of the reviewers and the process starts over again.

This process requires to keep track of the changes introduced to the artifacts and previous feedback, which together form a review history. To support this, most of the source code review tools provide visualizations of differences among the changed artifacts and additional annotations, such as comments tied to a specific scope of the text data. However, to the best of our knowledge, these visualizations only support textual artifacts, such as source code. Apart from binary artifacts, this way of visualization is also a problem for artifacts that encode data with text, but are hard to read for humans, such as models and diagrams.

In Model-driven engineering, domain models are the most important artifacts of a software project. Such models are represented by a textual syntax or a graphical syntax. The previously mentioned visualizations are dedicated to show differences between textual artifacts and are hence suitable for models with a textual syntax. However, they do not support models using a graphical syntax, unless they can be encoded in text. Although the latter is technically possible, it is desirable to visualize the changes and annotations using the graphical representation, as this overcomes the burden of learning another representation and keeping them in sync. It is expected that complexity is decreased and comprehensibility is increased by providing a visualization on the same level as the user is used to work with.

The graphical syntax of a modeling language may differ from modeling language to modeling language and is in general not constrained by any rules. Moreover, such a syntax is usually defined by actual implementations, as no widely accepted modeling language for such syntaxes exists to the best of our knowing. This makes it hard to find a single visualization of model changes and review annotations which works for all kind of possible graphical syntaxes, if any exists. Instead, it demands for an adaptable and extensible visualization method. However, if a visualization makes use of the graphical syntax, some conditions must be fulfilled in order to prevent misinterpretations by the user. The most important condition is that the two versions of the model must be easily identifiable. Also, it must be possible to distinguish between the model elements and the visualization of model changes and review annotations at any time for arbitrary graphical syntaxes. Additionally, the visualization must not modify the graphical representation of the model in a way that layout constraints of the graphical syntax are violated.

Besides that, a visualization for review purposes also poses the following *challenges*: Reviewing requires comparison of a usually well known model with another model, so it is essential to *easily identify changes as well as common parts of both models*. Preserving the so called Mental map [DG02, ELMS91, MELS95] with an appropriate visualization technique might support the reviewer during review. For example, exploiting the spatial memory of the reviewer in this way might reduce the burden of identifying common parts of both models. However, this must be done with respect to layout constraints, as mentioned before.

Additionally, models exist in various sizes and a submitted change may incorporate multiple smaller and also independent changes. So the *scalability* of the visualization in terms of the number of changes, as well as in terms of the size of each or both models is an important aspect for such visualizations. This demands a visualization technique that supports the main tasks *Overview*, *Zoom*, *Filter* and *Details-on-demand* of the visual information seeking mantra [Shn96].

Another challenge is the *visualization of annotations* defined by the reviewer, forming a feedback for the contributor. In current code review tools [Goo17b, Pha17, Bea17, Rho17, Atl17b] this is done by assigning simple text messages to parts of the code. Usually three types of code parts can be annotated this way. A sequence of characters, a sequence of lines or the whole file. In case of diagrams, which are instances of a modeling language

in a graphical syntax, the same should be possible with the exception that the parts are model elements or individual shapes of the graphical syntax. However, these types of annotations are very simple, related to only one element and usually unstructured. So it is not explicitly possible to specify relationships between comments that may ease the understandability of the feedback. This is not a big problem for textual artifacts as a typical reading direction can be assumed, and the reviewer may solve this issue by adding labels in the messages and referencing them in subsequent messages. In contrast, no typical reading direction can be assumed in diagrams. Although the workaround of labels may be used, a new problem arises. A referenced label might not be read yet, and the label must be found in one of the other messages, resulting in an additional search task that potentially distracts the user from understanding the content of the feedback. Therefore, depending on the size of the change and the extent of the necessary feedback, it might be of use to specify an order in which the annotations should be read, or to group annotations that address the same issue. Also the possibility of assigning multiple parts of the diagram to one comment could be of use in that case.

The last challenge is the proper *Visualization of the review history*. In a review process a reviewer is usually assigned to more than one review, multiple reviewers might get assigned to different contributions for the same review and the period between contribution submissions may be longer than several hours. So it would be of value for the contributor that the visualization allows to keep track of changes and annotations of formerly rejected submissions. This includes, but is not limited to the last rejected contribution, adding an additional dimension to visualize.

1.2 Aim of the Thesis

The aim of this thesis is to develop a set of visualization techniques to visualize graphical models in the context of review processes. Due to the challenges of graphical syntaxes mentioned in Section 1.1, the visualization must be designed in a way that adaption and extension to more complex graphical syntaxes is possible. These visualization techniques must consist of:

- Visualization of changes between two or more versions of a model using the existing graphical syntax of the model language.
- Visualization of reviewer annotations and their relationships to one or more parts of the model.
- Visualization of reviewer annotations that suggest modifications or some sort of labels that emphasize parts of the model.
- Mechanism that allows organizing of annotations.
- Visualization of the history of changes between two or more versions of a model using the existing graphical syntax of the modeling language.

- Visualization of the feedback, which is shown to the contributor.

Apart from that, visualization of review artifacts, such as comments, other reviewer annotations, the change history, as well as differences, must not modify the graphical representation in such a way, that the original models are not distinguishable anymore or the layout constraints defined by the graphical syntax are not fulfilled. A prototype implementation of the resulting techniques and a case study to verify the results is also presented in this thesis. The prototype is built on top of popular open source frameworks and tools used in Model-driven engineering, such as Eclipse [Ecl17a], Papyrus [Ecl17f] and the Eclipse Modeling Framework [Gro17]. Besides that, the prototype has been released under the terms of the *Eclipse Public Licence* [Ecl17b] in a public git repository on the development platform *GitHub* [Git17a] to aid further research and development. The repository can be found at `https://github.com/theArchonius/mervin`. Some of the visualization techniques elaborated in this thesis may also be applicable in the context of Model versioning and Model evolution [BKL+12] in general.

1.3 Methodological Approach

The Methodological Approach follows the Guidelines of *Design Science* [HMPR04]. The first two guidelines are already discussed in the previous sections of this thesis. Section 1.2 describes the artifact according to *Guideline 1*, which is a new visualization method for reviews of graphical models. The problem relevance according to *Guideline 2* is described in Section 1.1. The next step is to evaluate previous research of visualization techniques that addresses comparison and annotation of graphical models for review purposes. Section 2 contain this evaluation in detail including an evaluation of visualizations of current code review tools. Based on this evaluation and further research on visualization of graphs and diagrams, new visualization techniques have been derived and developed in Section 3. These techniques have been implemented in a prototype which is discussed in detail in Section 4. This prototype has been tested in a case study to verify the technique and identify new problems and restrictions which is discussed in Section 5. A final discussion and reflection of the work is done in Section 6.

1.4 Structure of this Thesis

This chapter provided an overview of the problem and methodological approach. The following Chapter 2 provides an in depth analysis of the current state of the art in code review tools and visualization techniques for differences in models as well as graphical models and diagrams. It also defines a common review process, the artifacts of that process and the involved actors in that process. Also other related visualization techniques are mentioned that may be adapted for graphical models and diagrams.

Chapter 3 presents a proposed combined set of visualization techniques that aim to improve the code review process for graphical models based on the findings in Chapter 2.

This is followed by implementation details on how the proposed solution is implemented in a prototype called Mervin in Chapter 4. Also the encountered problems are described in more detail in this chapter. As mentioned before, an evaluation has been done with the implemented prototype. The setup and the results of that evaluation are discussed in Chapter 5. Afterwards, future work is presented in the final chapter alongside with the summary of this thesis.

CHAPTER 2

State of the Art

Various tools exist that support reviewing of source code, but to the best of our knowing, not for diagrams or models with graphical syntaxes. For the sake of simplicity, reviewing such diagrams or models with graphical syntaxes is called graphical model review in this thesis. The first section of this chapter summarizes some of the most popular tools to provide an overview of the review process and to identify techniques that need to be adapted for graphical model review. Also they help to clearly define the overall review process and the tasks that need to be supported during reviewing. This is summarized in Section 2.2. The remaining sections are dedicated to visualization techniques in current research and tools that aim to support the previously identified tasks. Differences between versions are one of the most important artifacts while reviewing, so visualization techniques addressing the differences between two versions will be discussed in Section 2.4. Another group of important artifacts are annotations, which are discussed in the next section. The last section is dedicated to the visualization of the history of artifacts while reviewing.

2.1 Existing Code Review Tools

As mentioned before a huge variety of tools exist that aid at supporting developers during the review of source code. This analysis focuses on tools that support the whole review process from the beginning, where the first proposed solution is submitted, until the decision is made to merge it into the actual software. The goal is to identify a common process, common tasks and the commonly used artifacts. Obviously, reviewing code is a complex task and there may be also other tools that provide features that help finding differences and help understanding the effects of some aspects of the new code. However, such tools usually focus only on one aspect of the whole review process and need to be combined with other tools to form a tool set that can be used for code review. As users have to switch between such tools, it is reasonable to assume that this increases

Tool	Version
Bitbucket	Cloud Software, accessed May 17, 2017 - Server: 5.0
Differential (Phabricator)	commit 6ecd6980a1420633d5e02d09f67262d9dc876089
Crucible	4.4
Gerrit	2.14-773-Gd9d2de7604
GitHub	Web Service, accessed May 17, 2017
GitLab	Enterprise Edition / gitlab.com v9.1
Kallithea	0.3.2
Patchwork	v2.0.0-rc1-2
ReviewBoard	3.0
RhodeCode	4.7.2
Rietveld	commit 80a51fa637f416b8e9f192f9394bed2aca5ea1c3
Understand	4.0 (Build 891)
Upsource	2017.1.1892

Table 2.1: The version numbers of the analyzed code review tools

the complexity of the review as well. So the decision was made to exclude such tools for that reason and due to the large variety of combinations of such tools. That aside, visualizations of excluded tools that might be useful for graphical model review are covered in Section 2.4

The analyzed code review tools can be divided in two categories: Standalone software that is dedicated to support only code reviews, like Gerrit [Goo17b]. And code review tools embedded in development platforms such as GitHub [Git17a]. The latter are usually tightly tied to their development platform and its supported Version Control Systems (VCSs), whereas tools from the former category may also support other VCS that provides a description of the differences. Almost all tools are web applications, which might be due to the fact that different persons have to collaborate at the same time. The only exception is the software *Understand* [Sci17], which has been included for special reasons as described in the subsection below. The current version at the time of this writing of each of the mentioned tool have been used for this analysis. Table 2.1 shows the exact versions of the analyzed tools.

Plugins available for other editors and Integrated Development Environment (IDE)s that use the specified tools are not analyzed as they are considered to implement at most a subset of the features provided by the actual tools. Each tool has been analyzed by the features mentioned in their manuals, documentation, or by manual inspection, if possible. The assignment process of the particular roles in the review has been ignored for this analysis as they are either implied or the result of the configuration defined by the tool administrator. Additionally, the presentation of automated tests that have been executed to verify the new versions have been ignored. This is due to the fact that the visualizations beyond simple result lists depend on the actual model, which would be beyond the scope of this thesis.

2.1.1 Standalone Code Review Tools

Rietveld is an open source code review tool inspired by a code review tool used internally at Google [vR17]. This tool calls reviews *issues*. To start such an issue, an *issue owner* uploads the differences in so called *patch sets* using either the preferred upload script or the web fronted of the tool. Afterwards one or more reviewers may comment on the proposed changes. Issue owners may upload new patch sets and respond to these comments to discuss issues. They also have the responsibility to close the issue. It is not strictly defined who is responsible to merge the code in the final software.

An issue is presented with an overview page which lists the issue description, the involved persons, an overview of the patch sets and a chronological list of all comments. The patch set overview lists all changed files and their difference statistics. The differences per file are shown by default in the uploaded, color coded common unified diff format provided by the issue owner. This format is a merged version with all added and deleted lines with markers which define which lines have been deleted or added [Fre17]. Apart from that, the user may switch to a side by side difference view that allows the selection of the comparison scope with all uploaded patch sets and the original version. Reviewers may comment on the changed lines in patch sets and these comments will be shown below the specified line. Each comment is collapsible to show the complete comment text only on demand.

Gerrit is a popular code review tool used by several open source projects [Goo17a, Goo17b]. Gerrit is tightly coupled to Git, as it uses git commands and the internal object database of git to store parts of the review data. It was formerly created as a fork of the Code review tool Rietveld and therefore has some similarities to Rietveld. It also manages multiple *patch sets* with one or more *patches* per review. In contrast to Rietveld, a review is called a *change*, and most of its data is stored directly in the repository. A single contributor may create a new change by pushing a commit to a special branch on the gerrit server. A new patch set for an existing change can be uploaded by amending the uploaded commit, adding the associated change identifier and pushing it back to the special branch. Reviewers may add comments on a patch set or a sequence of characters in a patch. Contributors may also reply to them or write comments on their own.

Reviewers are also able vote for approval or rejection on a numerical range from minus 2 to 2 by default. A vote for minus two or two represents a strict rejection or approval, whereas minus 1 and 1 represent a not so strict rejection or approval. A vote for 0 does not have any effect. Gerrit also provides options to directly integrate the changes into the main development branch if an accept vote has been given. However, project owners may also define other *labels* that can be voted for as part of the review.

Access rights may be defined for users, groups and projects to control who is responsible for which task and allow the definition of flexible roles. The change overview screen is also similar to rietveld, with the following exceptions: Stacked bar charts are used to depict the file change statistics, showing the amount of added and deleted lines. The

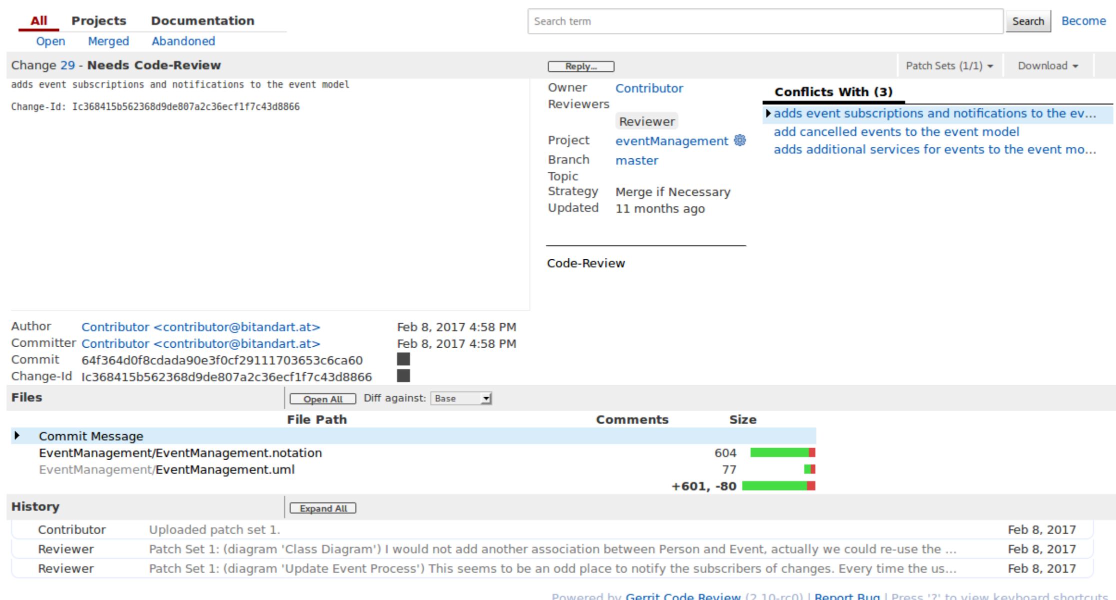


Figure 2.1: Screenshot of the change overview page in Gerrit.

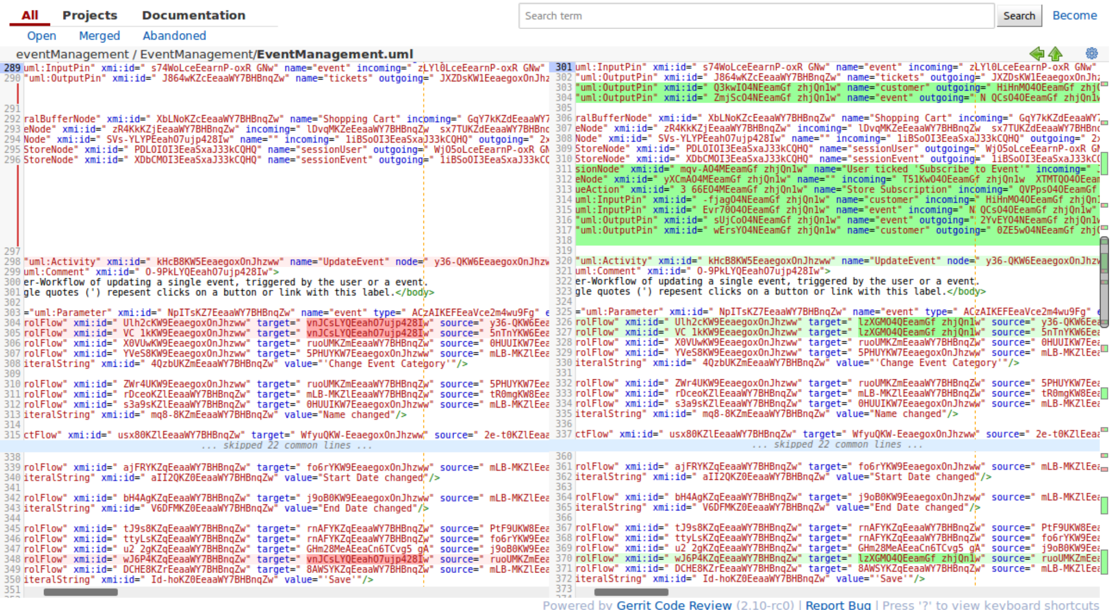


Figure 2.2: Screenshot of a difference visualization in Gerrit.

comment list also shows events and actions like the upload of new patch sets and votes. Conflicting and related open *changes* on the same Gerrit server are also listed.

Differences can be shown per file, in either a plain unified or a side by side view. Additions and deletions are color coded by their respective type. Only the context of the changed lines is shown by default, but it can be expanded on demand. Comments are shown after the line of the last linked character and the corresponding sequence is highlighted. Each of those comments are also collapsible, similar to the comments in Rietveld.

ReviewBoard is a code review tool that supports both reviews based on difference information upload or from a VCS repository [Bea17]. A script is also provided that simplifies the upload of the difference information. The submitted difference information is called a *revision*, and submitters may add new revisions for a review request. A review is called a *review request* wich is created by a *Submitter*. Multiple *reviewers* can comment on the request, on the uploaded difference information as well as on additional attached files. Moreover, comments can be marked as *issues* that have to be solved and will also show up in the the overview of the review request. Submitters may respond to these comments, close an issue, drop an issue or upload a new revision. It is also noteworthy that only the submitter is responsible for closing the review request by either marking it as discarded or submitted. However, this does not necessarily mean that the submitter is also responsible for integrating the difference in the final software.

The overview page of a review request shows the description of the request, followed by the issue list and finally the complete history of the review. This history includes all comments and all actions taken for this request. Revisions are shown in this history with a list of all changed files and a small donut chart showing the difference amount, color coded by their type. ReviewBoard distinguishes three types of differences: *additions*, *deletions* and *modifications*. Details of the revisions are only shown on request. All differences are shown on a single page in a side by side view. Each view includes the changed lines and some lines before and after to provide more context. It is also possible to increase the number of context lines on demand. Changed characters are color coded according to their change type. Additions and deletions that represent a *move* are also detected and marked with a reference to jump to the corresponding counterpart. Users may also restrict the displayed differences to a sequence of consecutive revisions, showing only differences between the selected revisions. Comments can be added for a sequence of lines for text files, or on text blocks if the changed file is a markdown file. Each comment is shown as a marker on the right of the corresponding line. These markers are stacked if comments overlap and a number on the marker indicates the number of stacked comment markers. Hovering over the marker reveals the comments and their replies.

In contrast to the other tools, difference visualization of images and comments on images are also supported. The simplest difference visualization is a side by side view of both images without any additional markers. The second option is to show the *difference image*. A new image that is created by subtracting the RGB values of each pixel of the two image versions. Another supported option is place both images above each other

and let the visibility of each be determined by the position of a slider. One version is visible on the left side of the slider, the other on the right side of the slider. The last supported option is similar to the previous one. Instead of controlling the visibility in a binary manner, the slider determines the blend factor of the two images. So if the slider is on the left or right side only the corresponding image version is shown. However, if the slider is in the middle, both images will appear semi-transparent on top of each other. Comments can be assigned to rectangular regions of the image. The regions are marked with an rectangle in the images and the assigned comments are shown when the user clicks on the rectangle.

Crucible is a standalone code review tool that supports the integration in existing development platforms [Atl17b]. Therefore it supports the creation of reviews from difference information extracted from patch files, VCS repositories and arbitrary files. Crucible distinguishes four review roles, where a single person may take more than one role in a review: Although the first role is named *author*, this person does not necessarily be the author of the proposed changes. According to the documentation of the tool, users with this role are users that deal with the results of a review. The *reviewer* is the person that reviews the changes and comments on the submitted code. As the name implies, *creators* are the users that create reviews. Users that act as *moderators* are responsible for assigning and notifying reviewers, and closing the review with a decision and a summary. Additionally moderators may also create reviews. All roles are allowed to comment or reply to comments. However, this is only the default recommended configuration which can be changed as the permissions of the roles and all users can be changed by the administrator.

Crucible also allows adding of new *revisions* or *changesets* to the review. The term revision is used for reviews based on patch files, while changesets are extracted from VCS repositories. The initial page of the review shows the current state, the description and the assigned reviewers with their progress on the review. General comments on the review are shown below. A navigation and a list of all changed files is shown on the right side with statistics about the amount of comments on the particular files. Comments can be made on the review itself, revision, changesets, single files or a sequence of lines. These comments also show up in the review activity list which shows all events related to the current review in chronological order. This activity feed can be shown on demand from any view of the review.

A detailed view of all differences for each file can be triggered by selecting a file in the file list. Crucible supports two different difference types: additions and deletions. Differences are color coded based on their difference type and can be shown in a unified view. A side by side view is also available on demand. Similar to all previously mentioned tools, lines with no changes will be hidden in these view except some lines before and after the changed lines. However, these hidden lines can be shown on demand. Filtering the differences is also possible for reviews with more than one revision or changeset. This can be done by selecting a sequential range of revisions or changesets to restrict the

displayed difference to the given range. Comments on lines are shown below the last line if it is visible. A comment associated to a hidden line is shown in the space that represents the skipped lines that contains the hidden line. The associated lines to a comment are highlighted with a different background color when the user selects or hovers over the comment. Users may also choose to mark their comment as issues or defects to indicate problems that need to be solved. Such comments are also accessible through the navigation list and may also be marked as resolved if the problem has been solved.

Patchwork is a code review tool that is primarily based on and controlled by e-mails or its Representational State Transfer (REST) Application Programming Interface (API) [Ker17]. Every information that a user submits, from simple comments and difference information to state changes, is usually parsed from emails from a mailing list. An optional script provides access to the REST API. The most important artifact that is parsed from this data is the so called *patch*. It contains the difference information, usually the content of a patch file generated by a VCS, and other metadata describing the patch. Additionally, *Cover letters* can be created that give an overview of a series of patches. Like in all previous tools, Patchwork also stores *comments*. These comments can be assigned either to patches or cover letters. Patches may have different configurable states, but Patchwork uses the states new, accepted and rejected by default.

Multiple patches can be grouped into *series*, *bundles* and todo lists. Users may also add new revisions of patches in a series, which results in the creation of new patches. Bundles are a collection of patches that can be created by the user. In contrast to a series, bundles may contain patches from multiple projects and patches may be contained in multiple bundles. A user may take one of two different roles: The standard user may submit patches for review, comment on reviews or create *todo* lists. *Maintainers* are able to change the state of a review, to archive a review, to delegate a review and to execute the same actions as standard users. Cover letters and patches are presented on a single page: The metadata is shown first, followed by the comments in chronological order. Depending if it is a collection or not, a list of the patches of all versions or the difference information is shown at the end. The difference information is presented in the submitted format with syntax highlighting.

Upsource is a code review tool that depends on VCS repositories [Jet17]. It is possible to create reviews either from one or more commits or from a branch. New revisions of branch reviews will be automatically added to the review, while new revisions of commit reviews must be added manually. All commits together form the final version that is under review. Upsource distinguishes three main actors of reviews: *Authors* are the authors of the commits assigned to the review, regardless of the user that actually created the review. One or more *reviewers* may give an opinion on the review by raising a concern or signal their acceptance for the new version. Additionally, *watchers* may also be assigned to the review. Any user that is involved in the review may close the review, adds comments to the review, or replies to existing comments. Reviews may be *pending*, *completed* or *closed*. A review is completed once all assigned reviewers made

their opionon on the review. The actual inclusion of the new version in the final software is the responsibility of any of the users and not necessarily managed by Upsource. It has to be noted that Upsource is one of the few tools in this analysis that supports code navigation and static code analysis for certain languages and projects.

The review overview page shows most of the review data without the detailed difference information. It includes the review title, the involved users, the revision list, the list of the changed files and the review timeline. Upsource tracks the number of viewed files per reviewer and shows them on demand in the user list of this page, to give an overview of the current review progress. The file list includes the number of added and deleted lines for each files. Every action of any user assigned to this review is listed in chronological order in the timeline. Comments may be added on the review itself, a line in the code or on a sequence of characters. The detailed differences are shown by default in a unified difference view. It is also possible to switch to a side by side view from the unified view. Each view shows the changed lines with unchanged lines before and after them for each file. Upsource recognizes added and deleted lines and presents them in different colors. The presentation of comments depends on the view: Line comments are shown directly after the line in the unified difference view and comments on a sequence of characters in a single line are not shown in this view type. Additionally, a line with a comment is highlighted with a colored background. Side by side difference views show comments on demand in a separated list view. To view the assigned characters or lines, users have to click on a comment. This enables highlighting of the assigned characters or lines with a colored background. Users may also restrict the scope of displayed differences by selecting the revisions to include. As revision might include different versions of the same line, this may also remove displayed code with assigned comments. Such comments are shown below the first visible preceding line with a special note. Comments can be marked with different labels to highlight special issues and each user may also mark comments as resolved. Resolved comments are shown like normal comments except that they have a resolved mark on it and its text is grayed out.

Understand is a static code analysis application that has some basic features that could be used in combination for code reviews [Sci17]. However, it does not define specific user roles or a particular workflow for the code review process, but can be used for the process without using other analysis tools. This is the reason why this tool is mentioned in this analysis but not in the same detail as the other tools.

Understand provides two main features that all of the aforementioned tools support: First, it supports pointing out problematic code fragments adding so called *annotations* in the code. Annotations are short texts with a particular author, a modification date and a list of key value pairs. Each annotation can be assigned to a file, line, or code entity. An explicit reply to such an annotation is not supported. Each annotation is stored in a database file that can be exchanged with other users. Apart from that, Understand also supports the comparison of two versions of code or text files.

Each version of the code must be checked out manually by the user as Understand does

not support loading versions form a VCS repository. A list of all changed entities in the comparison is shown at the top of the comparison view, including a control to filter the list. It is followed by a side by side view of the files with the highlighted differences. Differences are not distinguished in different difference types, like most of the other analyzed tools do. Understand differentiates differences by highlighting the currently selected, merged and not merged differences differently. Each side shows the code with or without hiding most unchanged lines, and parts of the code may also be collapsed or expanded, depending on the language entities. Additionally, the position of differences are also highlighted directly on the scrollbar. The differences are also shown in the common patch file diff format [Fre17] below the side by side view, followed by a list of all line differences.

2.1.2 Embedded Code Review Tools

GitHub is a web-based development platform based on git [Git17a]. It allows users to fork any accessible repository on this platform and change the code of the fork. Once the changes are complete, users may request the integration and review of these changes by creating a so called *pull request*. A pull request represents a number of incremental commits on a branch that should be merged into the target repository. This also means that multiple persons can be involved in the creation of the submitted changes. As most Git users know, the person that creates a commit is not necessarily also the author of the files in the commit. Furthermore, each commit can have different *commiters* and *authors*. After the creation of the request, one or more reviewers are each able to start a review of the changes by viewing the differences and adding comments. Once all comments have been made, the reviewer has to conclude the review with a summary and one of three different options: The most obvious options are to approve the changes or to request additional changes. Finally, the last option to conclude a review is to neither approve the changes nor request additional changes, but leave some feedback on the code. In case of the approval of the changes, any user that has permission to merge the changes may trigger the merge of the changes in the main repository. The last option to conclude a review is to neither approve the changes nor request additional changes, but leave some feedback on the code. How many approvals must be gathered or how many change requests must be incorporated to merge the changes, is usually the decision of the user that merges the changes. Still, GitHub may also be configured to accept merges only with at least one review that approves the changes. New commits can be added to the pull request by adding them to the branch. Users may add new comments to lines, files, and the whole pull request. Anyone involved in the pull request may also reply to these comments or add their own comments.

The pull request page shows by default the title, description, involved users and the history of the request. Additionally, a small bar below the title of the pull request gives an overview of the number of added and deleted lines in all files. All actions, comments and reviews are shown in the history in chronological order from the oldest to the newest. Comments and reviews are shown completely with the assigned code, if a code line has

been assigned. The only exception to this behavior are comments assigned to code that is no longer part of the pull request due to an update. Such code and its assigned comment are considered outdated and will be hidden unless the user decides otherwise. Commits are shown only with their author, the first line of the commit message and the commit id. Detailed difference information is only shown on demand by selecting the commit or the changed files tab. This tab reveals all changed files in the pull request in either a unified view or in a side by side view. Changed lines are colored based on their type and shown with some unchanged lines before and after them. Remaining unchanged lines are hidden unless they are explicitly requested by the user. Hidden lines are revealed in short steps, revealing only a few hidden lines per request. Comments on lines are shown after the line, with all replies below. All differences of each file are grouped. Each header of this groups contains a stacked bar chart showing the number of additions and deletions. Users are able to filter the displayed differences by selecting a sequence of consecutive commits to show. Comments on hidden lines did not appear in this view during this analysis.

Bitbucket is similar to GitHub, a web-based development platform that supports Git and Mercurial VCS repositories [Atl17a]. It is divided in two main products, the cloud service and the standalone server. Their feature sets differ from each other, but the review workflow and visualizations are mostly the same. So everything mentioned in this part applies to both versions if it is not stated otherwise. Bitbucket also uses *forked* repositories and *pull requests*, similar as it has been described for GitHub in the preceding paragraphs. But unlike GitHub, it does not require the explicit creation of a review. Users can signal their approval or remove it at any time. If the request is accepted or not, depends on the configuration of the repository: It can be configured to require an approval through a reviewer, but it does not have to. A user with the appropriate permission can accept the request without approval through a reviewer. Besides reviewing and contributing to pull request, users are also able to subscribe to pull request, to get informed of any action performed on the pull request. Feedback can be given with comments, and other users are able to reply to a comment. Comments can be assigned to a code line, a file or the pull request itself. It is also possible to create a list of tasks in a comment. These tasks can be accessed at any time by a keyboard shortcut or from the pull request overview page. Each task can be marked as resolved or unresolved while discussing or changing the pull request.

The main page of a pull requests shows the title, involved users, the description, and all pull request comments. At this point the cloud and the server version have a different layout. *Bitbucket Server* also includes the activity history in the overview tab, while the *Bitbucket Cloud* version show it in another tab on demand. On the other side, *Bitbucket Cloud* shows a section containing the difference information in the overview tab, while *Bitbucket Server* shows the this section in another tab on demand. The activity list shows a chronological list of all actions performed by any user on the pull request. Comments on lines are shown completely with a unified view of the corresponding line, including all replies. A comment will be marked as outdated if it refers to an line that is not part of the pull request any more. The last tab can be used to display all commits of the pull

request and show their contents. Although it is possible to comment on these commits, such comments will not be part of the pull request. As a result, they are considered not relevant for the review of the pull request.

The difference section starts in the cloud version with a short overview of the differences with a list of all changed files. This list includes the number of added and deleted lines, the change type of the file, and the number of comments related to the particular file. In the server version, the file list is shown as a file tree with the file color-coded based on their difference type. Files are categorized in *added*, *deleted* or *modified* files, while line differences are categorized in *added* or *deleted* lines. The overview is followed by the detailed differences in a unified difference view grouped by their containing file. Each changed line is shown with a few surrounding lines to provide context information. They are also color coded and marked with a symbol according to their difference type. Characters that do not match in a pair of consecutive deleted and added lines are more prominently highlighted than the other characters, pointing out that parts of a line have been modified. Unchanged hidden lines can be revealed in short steps on demand. Comments on lines are shown right after the assigned line with all replies. Outdated comments are not shown in the difference view. A side by side view of the differences without comments can be shown on demand. This view displays modified lines with a gray background, except for the added or deleted parts. Their background color is based on the type, either red or green. Also, both files are shown without hiding the unchanged lines but with markers next to the scrollbar that represent changed lines. Filtering of differences is only available in the server version of Bitbucket. It provides the possibility to filter the differences to a specific commit, or commits that have not been reviewed by the reviewer.

GitLab is also a web-based development platform that supports Git [Git17b]. This analysis focuses on the Enterprise Edition of Gitlab, but also applies on the community edition as the review features of both versions are identical according to the feature comparison matrix. Code review is supported for so called *merge requests*. They are either created from forked repositories as described in the paragraphs about GitHub and Bitbucket, or from a different branch of a single repository to a protected branch. This also implies that a merge request may contain code form multiple authors and committers, as explained earlier for GitHub. Once the request has been created, users are able to create comments, discussions and give their approval. Accepting the request does not necessarily require the approval of one or more users, but repository administrators are able to configure the repository to do so. A push of new commits from a local branch to the branch that is the source of the merge request is interpreted as a new version. All commits from the first diverging commit of the two branches to the newest pushed commit are part of such a version. Comments can be made on the merge request or on a single line in the difference information of the submitted changed files. *Discussions* are special comments that may have explicit replies assigned to them. It is not mentioned in the manual but during this analysis only merge requests may have simple comments assigned without replies. All comments on lines of code have been discussions and the

button to switch between those types was not present when adding a comments to a code line. Additionally, a discussion can be marked as resolved or unresolved and repository administrators may configure the repository to disallow merging of requests with unresolved discussions.

Each merge request is shown on a single page with its basic information at the beginning and tabs with more details at the bottom. Basic information includes the title, the description, a summary of the current request state. Given and requested approval, the involved branches and final actions are part of this summary. A collapsible sidebar shows the involved persons, labels and other minor information on the right side. An overview about unresolved and resolved discussion is shown next to the tabs. The history of the request is shown in the default tab, titled *Discussions*. It includes all actions as well as all comments which are shown depending on their type. Discussions are shown together with all replies, independent from the actual creation time of the reply. Discussions assigned to a code line are shown below the line, embedded in a unified view of the differences around the line. Resolved discussions are shown just with minor informations about the involved persons and the creation time. Detailed informations about such a resolved discussion can be shown on demand. All commits of the latest version are shown in the second tab. The last tab shows the difference information in detail: First, an overview is given with the statistics of the change: The number of changed files and the actual number of changed lines for each change type are shown. Gitlab distinguishes between added and deleted lines. Each changed file is shown with its changed lines and their surrounding lines. Hidden lines may be displayed on demand, revealing only a few more lines at a time. Difference information is shown either mixed in a unified view, or in a side by side view. Changed lines are color coded by their type. A stronger highlight is given for characters wich do not match in two subsequent added and deleted line sections. In other words, the changed characters of a modified line have a stronger highlight. Discussions on lines are shown directly below the assigned line with all replies. Users can hide all comments of a file or a single comment. Comments hidden this way are not completely removed from the view. Instead, markers are created with the picture of the first comment author next to the lines. Hovering over such a marker reveals the pictures of all involved users and a click reveals the comment again. It is possible to filter the differences to differences between two versions or the base branch. This also completely hides comments on lines that are not assigned to that particular difference scope.

Differential is a web based code review tool wich is part of *Phabricator*, a suite of web based tools for software development [Pha17]. Reviews are created either by submitting the difference information with a script provided by the tool or by manually uploading the difference information. Differential cannot be used without installing the whole Phabricator suite, although the submitting process is similar to the approach that a couple of the standalone tools use to be independent from the used VCS. A review contains one or more revisions, each representing the set of changes to apply to the existing code. Only one *author* exists for a single review. Authors are able to upload new

revisions and reply to comments. One or more reviewers can be assigned to the review and are able comment on the differences, reply to comments, as well as give an opinion on the differences. The latter can by done by accepting the review, rejecting the review or suggesting modifications to the differences. Comments are always submitted in a group: A set of optional comments on a sequence of lines with an optional comment that is not assigned to a section of the code. Merging the differences in the actual software is possible with an experimental prototype directly from the tool, but can also be done manually by any involved person, depending on the actual workflow of the team. Either the review is closed manually, or the merge of the difference closes the reviews automatically in the end.

A review is presented on a single page, starting with the review title, description, involved persons and other minor metadata. It is followed by a section that shows the related repository and the automated test results. A chronologic history of all actions performed on this review is presented afterwards, followed by the actual revision content section. Comment groups are shown in the history in the following manner. The comment not assigned to a code section is shown first, if it exists. Afterwards, comments assigned to code sections are shown grouped by their containing file, together with a link to the the code section. Detailed information about the differences is presented in the revision content section. Three tabs give an overview of the differences at the top of this section. A list of all changed files is shown in the file tab. It contains the path and the name of the files as well as its change state. Differential distinguishes between *added*, *deleted* and *modified* files. The history tab shows all uploaded revisions submitted by the author and allows selection of the scope of the differences to show in the detail view. Other metadata like the actual contained commits of the difference information is shown in the last tab. Line differences are shown below the tabs in either a side by side view or in a unified view. Changed lines are color coded by their difference type, where different character sequences of subsequent added and deleted line sections are stronger highlighted. Each changed line section is shown with it surrounding unchanged lines, while the other lines hidden. Hidden lines can be shown on demand either completely all at once or just a few lines at a time. Comments assigned to code lines and their replies are shown directly after the last code line. The assigned code section is highlighted with a different background color while moving the mouse pointer over the comment. Comments on lines are tied to a specific revision, but are also shown in other revisions as so called *ghost* comments at a similar position. How the position is determined is not exactly described in the manual of the tool. Ghost comments are grayed out to indicate that the comment has been assigned to a different code section in the beginning.

RhodeCode is web based collaboration and development tool with built in code review features [Rho17]. It is divided in the community and the enterprise edition, where the latter includes more features then the former. This analysis is based on the enterprise edition as it also includes all features from the community edition. RhodeCode supports the VCS Git, Mercurial and Subversion and implements code reviews for individual commits or pull requests. Both of them can be in one of the following review

states: *not reviewed*, *approved*, *rejected* or *under review*. Commit changes have a single author extracted from the commit and more than one reviewer may review the commit. Additionally, the review state of a commit is determined by the last review. Changing the review state of a commit does not have any consequence defined by the tool, but the users may take action based on the state. Reviews of pull requests are similar to pull request reviews in GitHub or Bitbucket. A pull request is created from a forked repository by a user that acts as the pull request author. Each pull request may contain multiple versions with multiple commits of different authors. An update of the pull request with a new commit also creates a new version with the old commit and the new commits. One or more users review the changes, leave comments and vote for the review state. Comments can be assigned to the whole pull request or a single line. It is not possible to reply explicitly to another comment. Once all votes are received, the request is either accepted or rejected. RhodeCode allows the merging of the request directly from the tool or manually by the user. Afterwards the request has to be closed manually, or the merge triggers the closing of the request.

The presentation of both reviewed commits and pull requests is similar: A section with general information of the reviewed element is shown at the top, including the current review state and the involved users. It also contains a list of all versions for pull requests and controls to select the version comparison scope that determines the differences to display in the difference section. A list of all commits follows this section for pull requests. The difference section is present for both elements, and shows the changed lines grouped by their containing file. Each file group gives an overview of the number of added and deleted lines of the file. All differences of a file can be hidden on demand. Differences can be shown in either a side by side view or in a unified view. Changed lines are color coded by their change type, which is either *added* or *deleted*. Modified lines are interpreted as deleted and added with a similar content and changed parts of such lines are stronger highlighted than the matching parts. Changed lines are shown with parts of their surrounding lines, while the other unchanged lines are hidden. In contrast to most of the other tools, hidden lines cannot be shown on demand. Comments are shown right after the line they have been attached to, and will be shown in all versions. An icon that displays the comment on demand is shown for comments on lines that are not part of the shown differences. The last section contains the chronological history that includes all comments without assigned lines and all actions applied to the pull request or commit.

Kallithea is a web based collaboration and development tool based on Git and Mercurial [Sof17]. It has been created as a fork of an earlier version of RhodeCode and there is no major release version of Kallithea yet. The review process of pull requests has not been documented at the time of writing this thesis and it can only be assumed that it works exactly the same as in RhodeCode. For that reason, only confirmed aspects of Kallithea will be mentioned in the following paragraphs

Kallithea also supports reviews of pull requests and commits. Reviews are also presented similar to RhodeCode: The general information is shown on the top of the page, followed

by the difference section and the history. A version list or a control to change the version comparison scope has not been found. Moreover, it is not clear if users are able to update the commits of an existing pull request or have to create a new pull request for every update. The difference information section contains an overview section at the beginning and the changed lines grouped by their containing file. A list of all commits in the request is shown in the overview section, including the review state of the commit and the total number of comments for this commit. It is followed by the list of changed files with icons that represents the corresponding file change type and stacked bar charts that show the ratio of added and deleted lines for the particular file. Each file group shows the changed lines color coded with surrounding unchanged lines in a unified view. Users can choose to increase the number of surrounding lines in small steps. Kallithea also distinguishes between added and deleted lines, like RhodeCode. Furthermore, it also shows the same highlighting behavior for modified lines like RhodeCode. Comments assigned to lines are also shown right after the lines they are attached to. A side by side view of the changed file can be shown on demand, which shows all lines at once. The scrollbars of each side are linked and markers next to each scrollbar indicate changed lines on the corresponding side. The color coding of added lines differs from the color coding used in the unified view and deleted lines are additionally crossed out. All actions and comments on the pull request, except comments assigned to lines, are shown in the history in chronological order.

2.2 Review Process & Artifacts

As shown in the previous section, the process varies from tool to tool, but all of them also have some parts in common. This parts are considered as the key parts of the review process. Also the name of the roles varies from tool to tool, so the role names in the following do not necessarily match the role names chosen by the tools. However, the role names in this and all subsequent sections have been chosen to be as expressive as possible and close to the role names used in the tools.

Origin of all reviews is a *change* that is introduced to an existing set of *base artifacts* to fulfill a certain *task*. A change may be separated in parts, which have been created by one or more persons, the *authors* of the change. All except one tool support updating the change, which results in new versions of the changed artifacts. Each update may also depend on a different sets of base artifacts. It depends on the tool if the old versions of the change are also available once it has been updated.

At the beginning a person, the *review requester*, requests feedback about a change from one or more *reviewers* in form of *comments*, *approval votes*, *rejection votes*, or other *annotations* on the artifacts. Annotations on the artifacts are realized with a varying *granuality* and in different ways in the analyzed tools. Some tools assign annotations to artifacts in a particular version, artifacts represented in all versions or artifacts present in the comparison scope of two versions. It also affects the *validity* of annotations after updating the change. All tools support *discussions* about the feedback to clarify issues

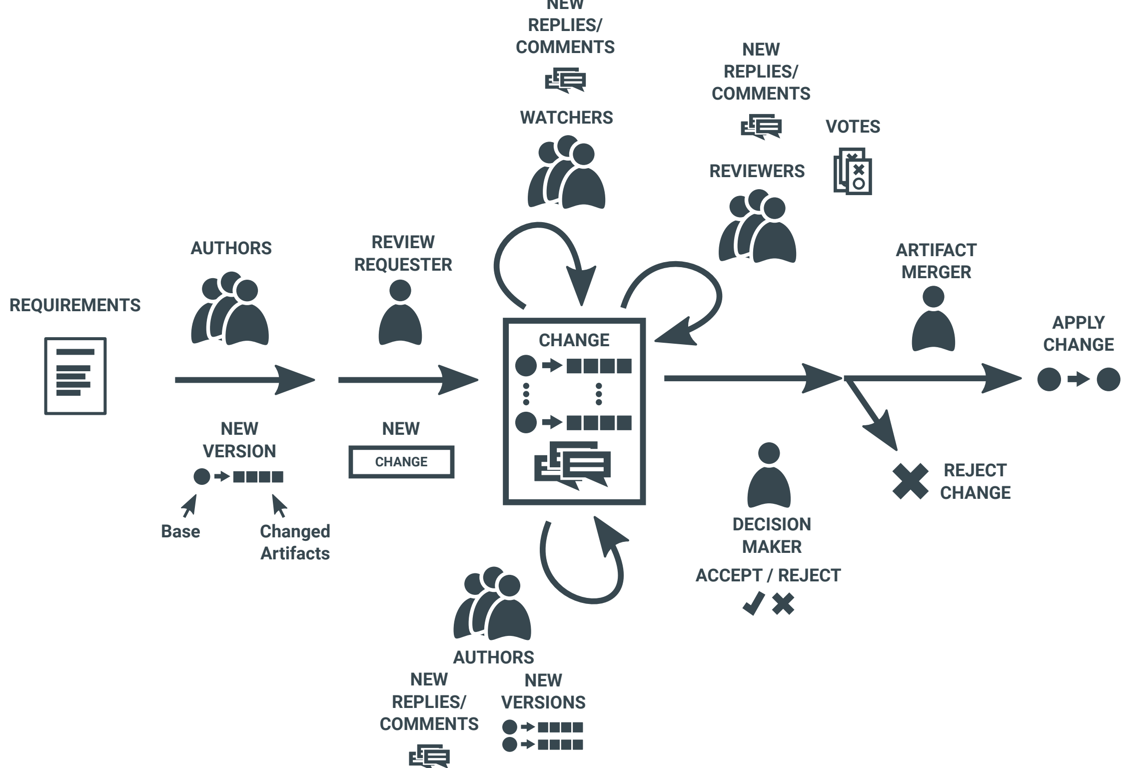


Figure 2.3: Illustration of the common review process and its artifacts.

and necessary actions to solve them.

A change might be updated until it is accepted or rejected by a person that has the permission to do so, the *decision maker*. It is the decision of the decision maker how many rejection or approval votes are necessary to make a final decision. The final incorporation of the changed artifacts in the set of base artifacts is the responsibility of the *artifact merger*. Additionally, some tools also support one or more *watchers* per change. Such persons are notified of actions taken by the involved persons in the process, and may also comment or annotate the artifacts. That aside, watchers cannot vote for approval or rejection.

Several different kinds of roles have been presented in the previous paragraphs, but not every review is done with at least one person for each role. Some roles like the watcher are optional, and a single person may take more than one role. For example a review requester may also be an author and may also be responsible for merging the approved change version. Any combination of roles is possible by the tools, and it is subject to the review policy of the team to define who is allowed to act in a particular role. Obviously, it can be assumed that a review cannot be done by a single person. This would contradict the idea of reviewing the changed artifacts and getting a second opinion by another person. So at least two persons have to be involved in a review.

Representations of a version in a change also varies: Some tools interpret versions as a single state of changed artifacts, like a single commit, which is seen as independent from the other versions. Gerrit is such a tool for example. On the other hand, some tools view versions as a sequence of applied sets of changed artifacts, the *change sets*, where each set might be contained in multiple versions. The final version is the final set of artifacts after applying the changed artifact sets in the order of the sequence. Bitbucket and other embedded tools that use pull requests or merge requests use this representation. Although this has no effect when viewing the differences between the base version and the new version, it is important once a user has to navigate through the version. For example, the user might be interested in the differences between two versions or filter the differences of change sets.

New versions are primarily viewed with focus on the differences with respect to another version. The main visualizations used to show these differences are unified or side by side views. Changed parts of the artifacts are shown with some context that show the point where the change has been applied. Some tools support filtering of differences on versions or change sets, but not on the change type. Changed artifact parts are highlighted color coded by their difference type. Almost all tools distinguish *added*, *deleted* and *unchanged* parts of artifacts with a certain difference granularity. In the case of the analyzed tools, the smallest possible difference was detected on a single line. Modified parts with respect to the smallest difference granularity are represented with an addition or deletion. Nevertheless, a couple of the analyzed tools also highlight the modified parts in a special way.

Difference types of artifacts are the same as the previously mentioned difference types, but also include *modified* artifacts. Most of the tools use red and green to encode additions and deletions, which may be a problem for users with color vision deficiency (CVD) [Sim10, JH06]. While this is the case, some of the tools also provide different color sets that can be chosen by the user on demand.

A study reveals that motivations for doing code reviews are *finding defects*, *code improvement*, *alternative solutions*, *knowledge transfer*, *team awareness*, *transparency*, and *sharing of code ownership* [BB13]. It also states that the key challenge is understanding the change and also the reason of the change. These challenges have been targeted by the analyzed tools in a similar fashion. Difference visualizations have been used to solve the first challenge, while textual descriptions are used to solve the second challenge. Hence there are some key aspects of the tools that have to be adapted for graphical model review: The various *change versions must be visualized*, especially the differences between versions, including the base version. Additionally, *filtering of differences* of one or more versions should be supported. Users have to be able *annotate the versions or differences* in some way to mark and discuss issues of a version. These annotations must be visualized in a way that allows to track back the *history of the review*, in order to check the validity of annotations with respect to updates to the change. As updates create new versions, an implicit history of changed artifacts or differences is also created. It is not explicitly visualized by the analyzed tools but might be of use for detecting

issues and argue about artifacts.

2.3 Graphical Models and Diagrams

Model-driven engineering is based the concept that "*Everything is a model*" [Bé05]. An arbitrary system is therefore represented by a model which conforms to a certain metamodel, and a metamodel may also conform to a metamodel [Bé05]. Models are more focused on the problem and closer to the problem domain [Sel03] than on the actual implementation with an actual technology. This allows to abstract the problem from the actually used technology to some extent [Bé05, Sch06, GS03, Sel03]. As a consequence, it focuses on the problem domain and therefore allows to describe complex problems while hiding implementation details. In model-driven engineering, models may also be processed. Currently, they can be executed, transformed, verified and used to generate actual executables or source code [Bé05, Sch06, GS03, Sel03]. Given that such generators exists for the target platforms and technology, models aim at offering the ability to describe and implement a system independently from both the target platform and the used technology.

Models and metamodels may be represented by a textual or a graphical syntax. Models or metamodels with a graphical syntax are called graphical models in this thesis since it focuses on this type of models. Models may also reference other models to form a single model, like it is done by the Graphical Modeling Framework (GMF) [Ecl17e]. This framework distinguishes the semantic model from the notation model which together represent a graphical model. The notation model references the semantic model and describes all graphical aspects of the referenced elements in the semantic model, to provide a clear separation between semantics and notation.

However, some artifacts do not encode the semantics of a model explicitly. For example images that store their data in pixels or vector data may depict graphical models, but do not provide semantic data without further processing. Following the previously mentioned approach of "*Everything is a Model*" also such images are models. But the domain is in this case to describe graphical data, not a model. So actually it is a model of a model [Bé05]. As the artifact is a model, it can be visualized like any other model, but it may be harder to understand for the user and more complex to obtain difference information for such artifacts. However it is generally possible to derive the necessary input mentioned in Section 3.8, which is required by the visualizations described in this thesis. To emphasize this, such artifacts are denoted as *Diagrams* in this thesis. The notation model without considering the semantic model is an example of such a diagram, since the semantic model is needed to obtain the actual semantics of the model. Diagram artifacts may be treated as graphical models if extraction of the depicted model is possible. However, such an extraction is out of the scope of this thesis. Hence, the combination of the notation model and the semantic model can be treated as a single graphical model.

2.4 Existing Difference Visualization Techniques

Change versions are displayed in all tools by highlighting the differences between versions with some context. Therefore, comparative visualizations of graphical models are needed and the following paragraphs describe current techniques and research in that area. The main focus in this section lies on comparison of two versions. Visualizations of multiple versions already form a sort of history and are therefore discussed in Section 2.6. Comparative visualizations of complex elements are difficult, especially in terms of scalability and complex relationships [GAW+11]. Diagrams and graphical models, as well as code are examples for such complex elements. It is even more complex in the case of code review or graphical model review as multiple versions have to be compared.

Although this thesis aims at finding a visualization that uses the graphical representation, difference visualizations of textual artifacts might provide hints and aspects that can be used also for other difference visualizations. Examples are the principle of side by side or unified visualizations [SR04], which have also been encountered in the analyzed code review tools.

Side by side visualizations of textual artifacts are based on *juxtaposition* and *explicit encoding* [GAW+11] of differences. One version of the text is shown next to the other side. Changed lines are highlighted according to their change type with a different background color and a corresponding symbol at the beginning of the line. Insertion or deletion points on either side are linked to their corresponding text block on the other side with their specific color. Unmatched characters of subsequent added and deleted text blocks are stronger highlighted to emphasize modified parts. Scrolling of long texts is usually done in a linked way, such that corresponding parts of both parts are shown at once. This may be implemented by adding blank space corresponding to the larger side of the changed text block on the other side, or by scrolling each side by a different amount. Visualizations that use the latter method also provide a scrollbar for each side as the scrolling amount is also dependent on the version to scroll. For example, a large text block on the left side that has been deleted on the right side result in a big amount of scrolling distance on the left side if the right side is scrolled just by a small amount. This potentially skips parts of the text block and makes it hard to scroll for the user. On the other hand, normal scrolling on the left side scrolls the right side just a small amount and the large text block can be read easily.

Unified text difference visualizations are based on superposition and explicit encoding [GAW+11]. Both versions are merged into a unified version that includes all changed lines. The highlighting and color coding is the same as described for side by side text difference visualizations.

Unchanged lines are usually hidden in both visualization types except for lines close to changed lines. They are shown to provide context to the user, and this context can be increased in most tools. A line overview visualization is used by most of the other tools to highlight changed lines and to support quick navigation through the text if no lines are hidden. Changed lines are marked on a vertical bar that represents the whole file.

Markers are color coded by their change type and positioned on the bar relative to their line position.

None of the analyzed tools supported displaying of differences in diagrams or graphical models. Despite this, diagrams and graphical models can also be seen as individual images that can be compared. On the other hand, not much research has been done explicitly in the field of visualizing differences of two images, to the best of our knowing. Obviously, the most simple ways to show differences of two images are to place them next to each other or to calculate the a difference value pixel per pixel to generate a difference map. The latter has been done by the tool *ReviewBoard* [Bea17] or by Baudrier and Riffaund [BR07]. Alternatively, different interactive blending approaches may also be used to allow interactive comparison, as described for the *ReviewBoard* [Bea17] tool.

EMF Compare [Ecl17c], the comparison framework for Eclipse Modeling Framework (EMF) based models provides a comparison editor for Eclipse based products. It also supports graphical comparison of graphical models based on Papyrus which itself is based on GMF and EMF. The editor is divided in three parts: the difference tree, the base version and a side by side view. EMF Compare creates a tree structure of matching model elements, assigned with the detected differences on the particular match. This tree is reduced to matches with differences, filtered, grouped and finally shown in the difference tree. Applied filters and the grouping algorithm can be selected by the user. The number of available filters depend on the installed plugins and the model. But the cascading differences filter should be mentioned at this point: It filters all differences from the tree that are contained by other differences. For example, only one addition instead of multiple additions remains, if an element is added that contains also child elements.

Three different difference types can be shown by the editor: *Additions*, *deletions*, and *changes*. It should be noted that the API of EMF Compare also distinguishes between *changes* and *moves*, which are mapped to one of the previously mentioned difference types. Additionally, conflicting differences and merge states are also visualized, but will be ignored in this thesis as the task of solving merge conflicts has not been observed to be supported by any of the analyzed code review tools during the review process. Instead, reviewers requested the authors to resolve the merge conflicts and update the change.

Each difference type is represented by a small icon before the difference label in the tree. A more detailed difference description is also added to the label. The content of the two remaining parts of the editor depend on which element of the difference tree has been selected. As the name implies, the base version shows the common base version of the compared element versions if it exists. It is completely hidden otherwise. The side by side view shows the versions next to each other and is also dependent on the selected tree element. Additions and deletions are detected with respect to the left side, which means that the content on the left side is considered as the new version. Modifications of non-containment features in EMF models are shown without context but with links to corresponding elements, insertion points, or deletion points. The same is done for

modifications of containment features, except that the whole containment tree is shown to provide context.

A graphical representation is shown for each version if a difference is selected that corresponds to a graphical element depicted in a Papyrus model. Each side view is centered to the selected element or the area affected by the difference. Changed elements are outlined, depending on the element. A node that can be freely positioned is outlined by a simple rectangle that shows the position of the node in both versions. If the node is not present in the version, an outline is positioned at the position where it has been or has been added in the other version. Edges are outlined by lines which follow the path of the edges, together with outlines of the source and target elements. Positioning is done in the same manner as for the outline of nodes. Elements presented as a sequential list contained in another element are simply outlined with a rectangle if they exist, or highlighted as a line representing the insertion point or deletion point in the list. Scrolling the content of a side does not affect the other side, whether it is a list, tree or the graphical representation of the difference.

The widely used side by side visualization for textual artifacts has been extended for graphical models by Schipper et al. [SFvH]. Based on the compare editor of *EMF Compare* [Ecl17c], they used an additional tree view that shows a structural overview of the differences. This tree view also supports focusing on particular differences by zooming the side by side view to the current selection, which allows interactive navigation through all differences. The side by side view encodes difference information with color and supports zooming and panning. To improve the scalability, the authors suggested to collapse regions which are considered to be not of interest, with respect to the graphical syntax. However, this does not improve the scalability for a large amount of differences and also introduces new challenges: The identification of these regions in arbitrary graphical models and how to collapse them without violating the graphical syntax.

Ohst et al. [OWKa, OWKb] proposed a technique that aims to provide a visualization of differences in UML diagrams and other design documents containing graphical diagrams. They focused on diagrams where the particular layout is considered irrelevant, so layout constraints of graphical syntaxes are ignored and not supported. With this restriction, both diagrams could be merged into a single diagram with a changed, but similar layout. Difference information is encoded using line styles and colors. Additionally, developers could select and restrict the encoding on specific logical parts of the difference which gives the developer a filtering mechanism.

A similar encoding has been chosen for a technique proposed by Mehra et al. [MGH]. In contrast to the previous method, they merge both versions into a single diagram without modifying the layout of the individual shapes. Also, they visualize changes of shape properties by outlining the previous shape and drawing a line to the new shape. However, due to the simple merging of both diagrams, it is quite likely that shapes of both versions or outlines overlap, hiding potentially important information.

Also, *Polymetric Views* can be used to visualize model differences [Wen]. In this approach

a set of change metrics are defined and calculated for specific entities of a model instance. Afterwards the values per entity are mapped onto properties of a rectangle, and relationships between entities are shown by lines between the corresponding rectangles. This provides an overview over even vast models, and changed entities can be identified at a glance. Details of a particular change are shown on demand by selection of entities. The authors also propose different sets of metrics that target different aspects of the change or aspects which depend on the chosen modeling language. Later work [vdBPV] refined this method and proposed an additional step that allows viewing the details of changes for a selected entity using the graphical syntax of the modeling language. Such a detailed view is a unified diagram of both versions restricted to a selected subset of the model. The difference information is encoded by using different colors for each difference type. However, a mapping to a specific *dot metamodel* must be defined which is used to create the unified view. Although this model is quite flexible it does not support preserving the layout information of the original model versions, and is therefore not capable of preserving the mental map of the user.

The differentiation tool DSMDiff [LGJ] visualizes changes in a non-graphical way by presenting the model instances with highlighted changes in a tree view. A similar approach is implemented by *EMF Compare* [Ecl17c], where tree views of the underlying models are presented side-by-side and changes are highlighted. Störrle [Stö] proposed an even simpler representation extracting high level changes in form of prosaic sentences or tables. Although these techniques are not embedded into the diagram or use the graphical syntax, they may serve to define additional views that support navigation, filtering and other interaction tasks in the context of visualization.

2.5 Element Annotation Visualization Techniques

Annotations of artifacts are used by all of the analyzed tools and are one of the key aspects to formulate and discuss feedback on the changed artifacts. To the best of our knowing, no explicit research has been done in that area, so this section summarizes the different annotations encountered in the analyzed tools.

In the analyzed code review tools, comments assigned to code are the only way to annotate the artifacts. Comments usually contain at least a message and are always tied to a single artifact or part of the artifact. They are shown in three different ways: The first option is that the comment is shown right below the assigned element, moving all subsequent elements below the comment. Another way is that a marker is added next to the element and the comment is shown on demand, hiding parts of the subsequent elements in the artifact. The last option is that the comments are shown completely detached form the assigned elements, and selection of the comment triggers a highlight of the assigned element. The other two options may also be combined with a highlight of the assigned elements with a special color, either permanently or on demand, depending on the tool. *ReviewBoard* also supports comments on regions in images, and uses a rectangular marker to show the comments on demand. Some tools also support comments

that have one or more replies assigned to them, which are all displayed right below the replied comment.

Users are also able to hide the comments. Some tools remove the comments completely with all highlights from the view, while other tools replace them with markers to indicate that comments exist. Additionally, a couple of the tools also support marking the comment with labels to indicate issues that have to be solved. Such labels or issues can be manually removed or resolved by other users, which may also result, depending on the tool, in a grayed out representation of the comment. A few tools use the labels to generate overview lists of unresolved issues and requested changes.

Comments assigned to elements that have become invalid due to updates to the element are also differently displayed. The easiest but also used option is to do not display them at all. Another option is to show the comments grayed out at the nearest valid element. The last option is to show a hint that views the comment with the old elements on demand.

2.6 Review History Visualization Techniques

History data of a review includes all actions taken by users and also relations between artifacts and annotations in multiple versions. The presented tools usually presented this history to some extend in a chronological list, mostly with focus on the actions performed by users during the review process. But they also include comments and other annotations on the artifacts or differences. Excerpts of the related artifacts are also presented for annotations in the list. Of course only if they are part of the history list, as some tools do not include them in their history.

As mentioned before, also different versions of artifacts form a history for each artifact or a history for a set of artifacts. None of the tools visualized this type of history as they usually focused on a small part of the history: the difference between two versions. This might be due to the fact that the former adds another dimension to existing visualizations, as differences of multiple versions must be visualized.

To the best of our knowing, no research has been found on graphical model history visualizations that use the graphical representation of the model elements. Nevertheless, some visualizations exists that aim to visualize the evolution of code or artifacts in an abstract way. They might be also adopted for graphical models, due to their abstract representation of artifacts and their properties.

A couple of them focus on particular history metrics which are dependent on the actual type of the artifacts. These metric values are mapped onto elements or properties on a graph, or are used to animate elements or properties on a graph [FG04, CKN+03]. Others use static pixel mappings and animated visualizations to show the change in metrics or change metrics itself [BE94, BE96]. Another option is to visualize the evolution of artifacts with rectangles representing artifacts placed in a matrix [Lan01]. Rows represent

the artifact in the examined version range, columns represent a single version. If an artifact does not exist in a version, no rectangle is shown.

A similar, but more dense approach is to visualize file evolution in a matrix [VTvW05]. Lines in a file are represented by rows and columns represent versions. Color coded cells indicate if a line has been modified, added, deleted or not. Alternatively, color coding can be changed to show authors, line types or other line based metrics. Additional views show metrics with a higher granularity and the actual text that can be selected by the user by selecting lines and versions. Moreover, selecting empty lines in a version shows also lines that have been deleted or will be added in a later version. Coordinated views with a matrix-based visualization has also been proposed for visualizing rank differences in search results for multiple or revised sets of search terms [SR04]. Rows represent search result entries and columns the set of search terms. Cells contain color coded circles based on the rank of the entry in the result set of the corresponding set of search terms. The other views present overview and details of the dataset with respect to the current selection.

Multiple tree visualizations with edges that link common parts are also an option to visualize artifact evolution [CAT07, TA08, GK10]. As mentioned in Section 2.4, diagrams and graphical models can also be seen as images and multiple versions of images form a sequence of images. Sequences of images are also commonly found in videos, and a visualization has been proposed that aims at giving overview of videos [DC03]. These focus on visualizing differences between single images in videos, by creating volumes out of difference maps and hiding unchanged parts. This might be also applied for multiple versions of images created from diagram or graphical models, although it might need to be adopted for the potentially smaller amount of images.

CHAPTER 3

Mervin: A Graphical Model and Diagram Review Tool

The previous sections showed that no tool or set of visualization techniques exists that provides a set of visualizations that support the mentioned open problems. Hence, a set of visualizations is proposed in this chapter which have been implemented in a prototype called Mervin. This chapter explains the visualization techniques in general, which are independent from the actually used frameworks in the prototype. Implementation specifics of the prototype will be discussed in Chapter 4.

Many different aspects of reviews have been presented in the previous chapter that need to be visualized. A set of coordinated visualization views [NS00] are therefore proposed instead of a single visual representation for all these aspects. They use *brushing and linking*, where selections in one view leads to highlighting in the other views [NS00]. Additionally one view is also designed to use *drill-down* coordination, where the selection in one view yields the visualization of child or related elements of the selection in another view [NS00]. Understanding the change is the biggest challenge in code reviews [BB13], so it can be assumed that this also applies on graphical model review. Understanding the change also includes understanding the reason for the change. While this is the case, the reason is usually part of the textual description of the review request and is not necessarily explicitly represented in the submitted changed artifacts. Hence, the presented visualization focuses on helping understanding the changes of the submitted change versions.

Five views with different visualizations are defined to achieve this goal. Two of them focus on visualizing differences and annotations for a selected pair of versions, be it change versions or one of their base versions. They are referred to as the *old* and *new* version of the current *comparison scope*. Other comparison tools also refer to those as the *left* or *right* version, but this also requires the definition of an implicit or explicit reference

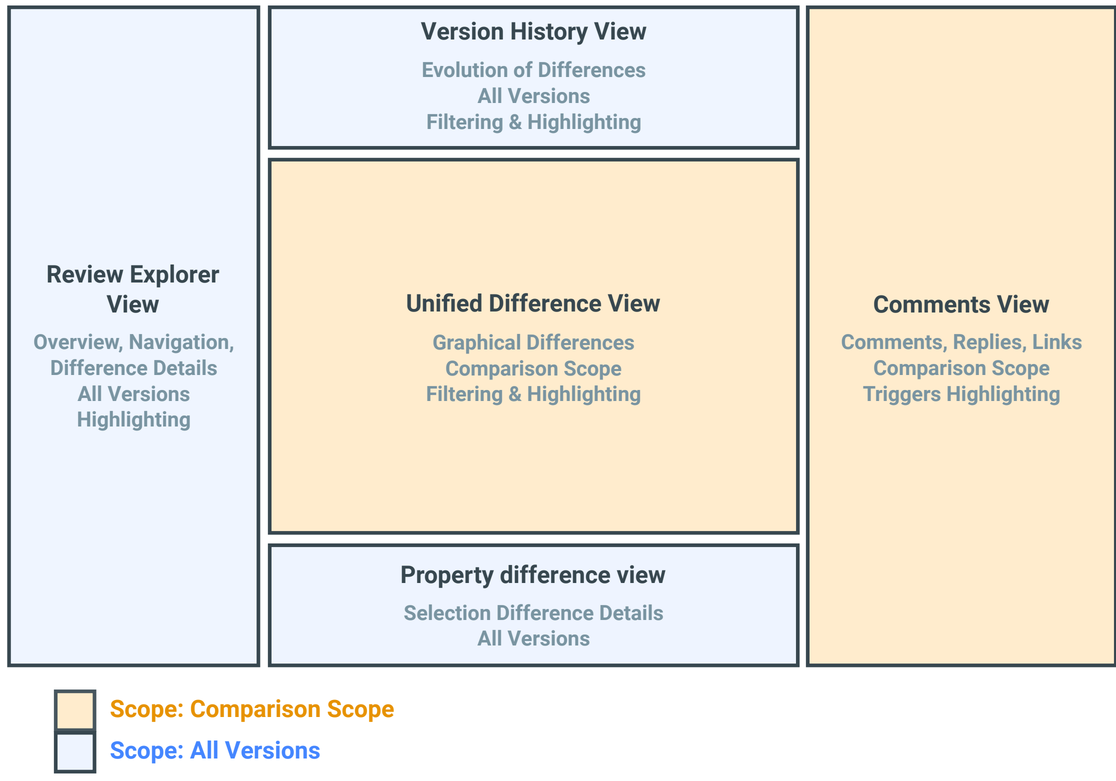


Figure 3.1: Illustration of the set of coordinated views in the proposed solution.

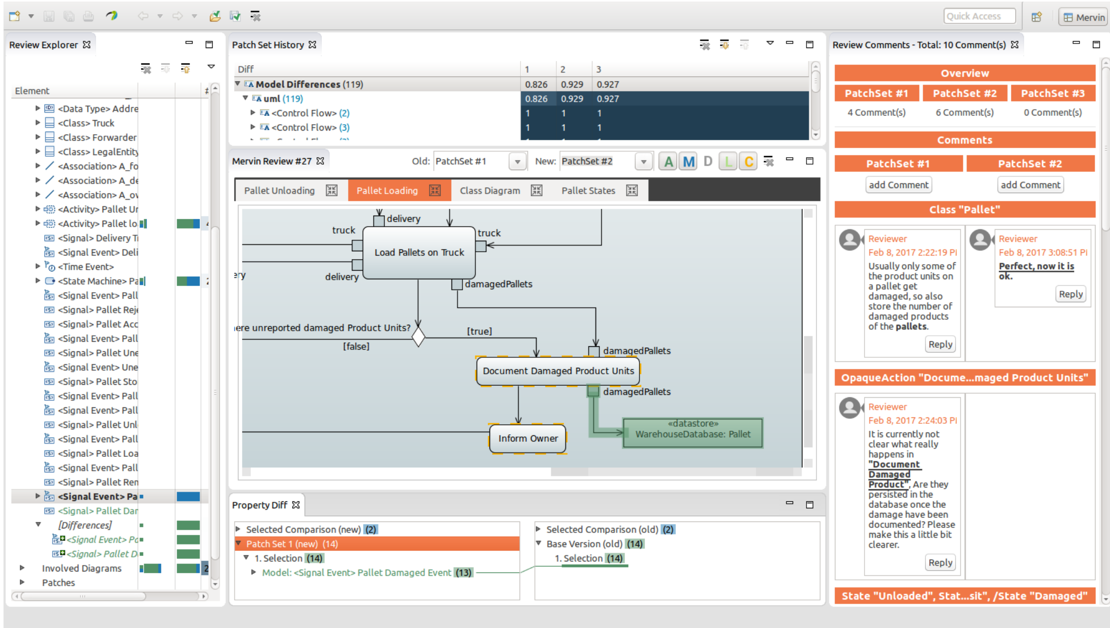


Figure 3.2: Screenshot of Mervin, the prototype that implements the proposed solution.

side to avoid ambiguity of differences. Otherwise, the classification of a difference as an addition or deletion is not clearly defined. The notion of an old and new version eliminates the need for such a reference side. Additionally, naming the versions based on particular sides also implies that the versions are shown in some sort of side by side view. This is also not true for the central difference view, so the new and old labels have been chosen.

The layout of the view is based on the concept of one central view that takes the most part of the screen, surrounded by supporting views. Controls to change the comparison scope and the difference visualization of the selected comparison scope are part of the central view, the *unified difference view*. Comments are shown right to the central view, in the *comments view*, which also depends on the selected comparison scope. On the left side of the central view is the so called *review explorer view*, which aims at providing an overview and navigation through the versions of the review. The top view is the *version history view* which aims at visualizing the history of versions in the review. Below the central view is the *property difference view*, that shows fine grained difference details based on the current selection. As mentioned before, the review explorer view, the version history view, and the property difference view also visualize aspects that are not part of the current comparison scope. All views show different aspects of the differences, and therefore address the problem of understanding the change. Understanding the reason of the change and the outcome of the change is not addressed by any of the views, like in all of the analyzed code review tools. Supporting this aspect may be addressed in further research.

Positions of the views have been chosen to support the expected user interactions. The central view is the most prominent view and is expected to draw the most attention of the user at the beginning of the review. By default, the comparison scope is set to the base version of the latest change version for both the left and the new version, which results in showing the base version without differences. The user may use that to familiarize with the models before proceeding with the review, if necessary. Afterwards, the user can set the comparison scope or start searching for issues using unified difference view, the review explorer, the version history or the comment view. Quick switching between views is expected to be encouraged this way, so that potential issues can be confirmed in combination with the other views. So almost all views act as the starting point for investigations of the user. Only the property difference view is designed as a detail only visualization.

The views are also designed to address the problem of *scalability*: Diagrams and graphical models vary in size and therefore also take up a varying amount of space. Obviously, screen space is limited by the actual display devices and must be shared by all views. So parts of the views must be hidden if there is not enough screen space and potential important information is not shown to the user at once. It is also expected that not every view is used in the same frequency by every user for every task. Some tasks do not even require the use of a certain view and the screen space can be used for a view that is more important for the user's needs at a certain point. To reduce the problem of

scalability, each view may be resized or temporarily hidden on demand by the user.

3.1 Difference Types

Visualizing differences between versions, including base versions, are an integral part of code review and will also very likely be an integral part of graphical model review as stated in Section 2.2. It is therefore necessary to define the difference types that may be encountered in graphical model review.

The most obvious difference types are based on low-level *atomic operations* [BKL+12] applied to the model. *Additions* and *deletions* represent new model elements and deleted model elements. EMF Compare also distinguishes a *change* from additions and deletions, representing elements that have been moved between different features and modifications of special features. This difference type is also used in the proposed visualization and is called a *modification*, to avoid ambiguity with a change in the review process. *Layout differences* are another category of differences. They represent differences in the layout of the concrete graphical syntax, including *location* differences, *dimension* differences, and *edge routing* differences.

A location difference represents the move of a model element from one position to another position. Each position is relative to a reference point, model elements might be nested in parent model element, and might also be moved between different parents. A dimension difference represents a modification of the extent of the graphical representation of a model element. That might be the width or the height of a rectangle or any other shape that is used to describe the bounds of a graphical representation. Edge routing differences are differences of the layout information of an edge or connection. They describe differences of the start point, the end point, or the exact route that the edge takes. Such a description can be a list of points relative to a reference point or any other set of parameters that can be used to define a curve.

Layout information may also be implicitly or explicitly defined. For example, the width of an element may be defined as the width of its biggest nested element. So a layout property depends on one or more properties of another graphical representation of an element and is therefore implicitly defined. Additionally, the implicit dependency might be replaced by an explicit layout information or vice versa from one version to the other. For static graphical representations, explicit layout information can be computed from implicit layout information, given that the concrete layout algorithms are known. In this case no implicit layout information differences are needed in theory, but it remains to future work if this has any effect on understanding layout differences more easily. The following sections also describe the handling of implicit layout changes.

Layout differences can be derived from atomic operation differences, depending on the detection of such differences. No restrictions are made in this thesis how these differences are computed exactly as it depends on the underlying modeling and difference

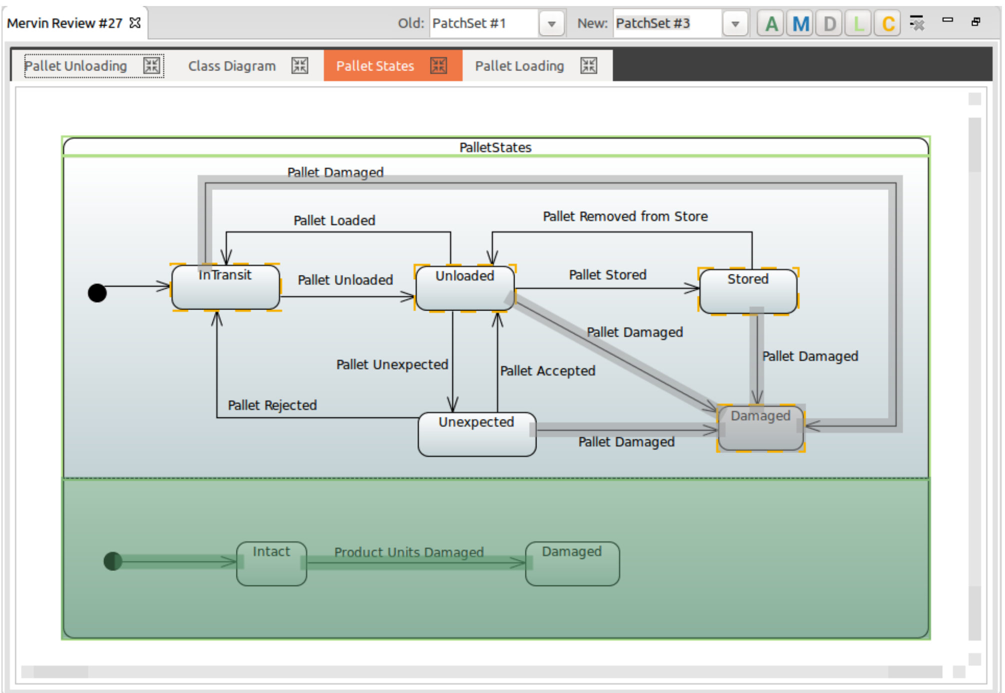


Figure 3.3: Screenshot of the unified difference view in Mervin, the prototype that implements the proposed solution.

computation framework. While this is the case, Section 4.5 provides an example on how to compute such differences based on Papyrus and EMF Compare.

Layout differences are sometimes considered to be irrelevant to the semantics of a model [OWKb]. Even tough this is true for some models, layout issues may be a valid reason to reject a change version. In that case, a visualization of layout differences in an updated version may support the understanding of the layout differences, although the semantics of the model have not changed.

3.2 Unified Difference View

The unified difference view shows differences in diagrams and graphical models with their graphical representation or using their concrete graphical syntax. Both are build up of graphical elements and the supported structure of graphical elements is described in the following paragraphs. It consists of a few basic elements, *nodes*, *anchor points*, and *edges*.

Nodes may contain multiple nested elements, and an element can only be contained by a single node. So nodes and their nested nodes actually form a tree of nodes. Not all elements are actually visible to the user, but they are used describe the composition of the graphical elements. Nodes may have an arbitrary shape, but their bounds can be described by a simple shape like an rectangle that encloses the shape. This rectangular

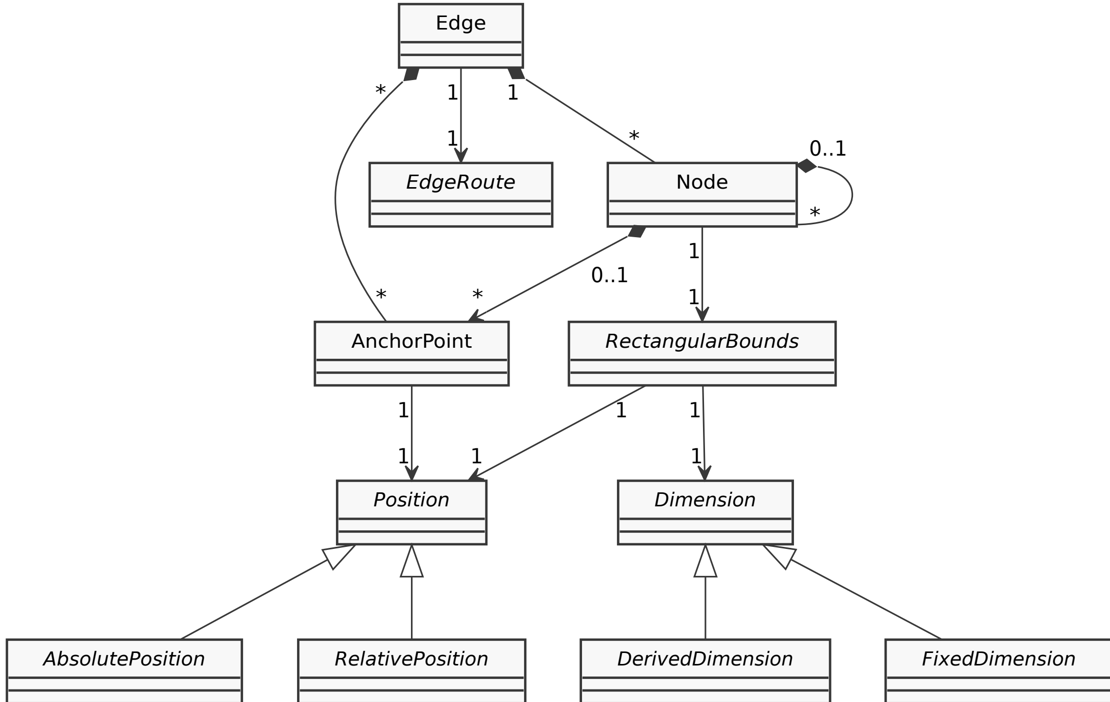


Figure 3.4: Outline of the used structure of graphical elements within a diagram or graphical model.

description of the bounds is the basis for the visualizations and differences of nodes in this view. The proposed solution focuses on two dimensional graphical representations, but a cuboid can be used instead if the graphical elements are shown in a three dimensional space instead of a two dimensional space. However, both options also require the definition of a *position* and a *dimension*. A position may be *relative* or *absolute*. Relative positions are defined to be relative to some reference point which is also at a certain position in the view space. An absolute position is defined by coordinates that are relative to a reference point of the view, usually the top left corner of the view. Similar, dimension properties may also be derived from other properties like the width of contained nodes, or fixed to a certain value.

Anchor points are usually invisible, and are assigned to one or more ends of one or more edges. They may be also assigned to other elements to link an edge and an element together. In that case, the position of an anchor point is often relative to the assigned element.

Edges are graphical elements that are used to connect exactly two anchor points. Again they can have any shape, but the shape follows a certain route from one anchor point to the other. Additionally, edges may also have child nodes or anchor points with positions relative to the edge. A label on an edge is an example for such a child node.

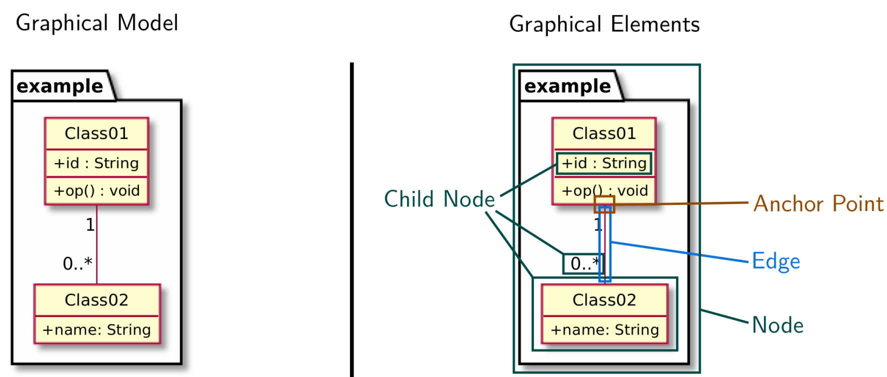


Figure 3.5: Example and visual explanation of the used structure of graphical elements within a diagram or graphical model.

This structure has been designed to support a wide variety of different graphical representations and should be sufficient for most diagrams or graphical models. For example, the simplest way to create such a structure for an arbitrary diagram or graphical model is to create the hierarchy of nodes based on containment relation of shapes. Graphical representations that contain edges that connect to elements can be obviously mapped the edges in the presented structure. Edges that connect multiple elements may also represented by multiple edges in the presented structure that share an invisible anchor point.

Given that structure, differences are shown in a unified manner in the unified difference view. In short, that means that two versions are shown in one merged version, with different highlights to point out the differences. It contains of the common parts and the changed parts of both versions [OWKb, OWKa]. As mentioned before, the versions are named the *old* and the *new* version of the current *comparison scope*. Added elements are from the new version, deleted elements are from the old version, and modified elements are contained in both versions. The comparison scope is selected by the user, and selecting the same version as the old and new version results in the display of the selected version without differences. Prior work on generating such a unified model [MGH, OWKb, OWKa] show that keeping the layout of changed model elements might result in overlapping elements. They suggest to modify the layout to avoid such overlaps. Nevertheless, computing a new layout might be complex and require detailed knowledge about the effects on the semantics of the model. In this solution, interaction is proposed as a partial solution to overcome the problem of overlapping elements. Hence, no explicit layout adjustment is done in this view and absolute as well as relative positions are kept. Deleted elements change the layout of the diagram depending on how derived dimensions and the reference points of relative positions are defined. For example if the reference point is defined to be the bottom left corner of the previous sibling element, also

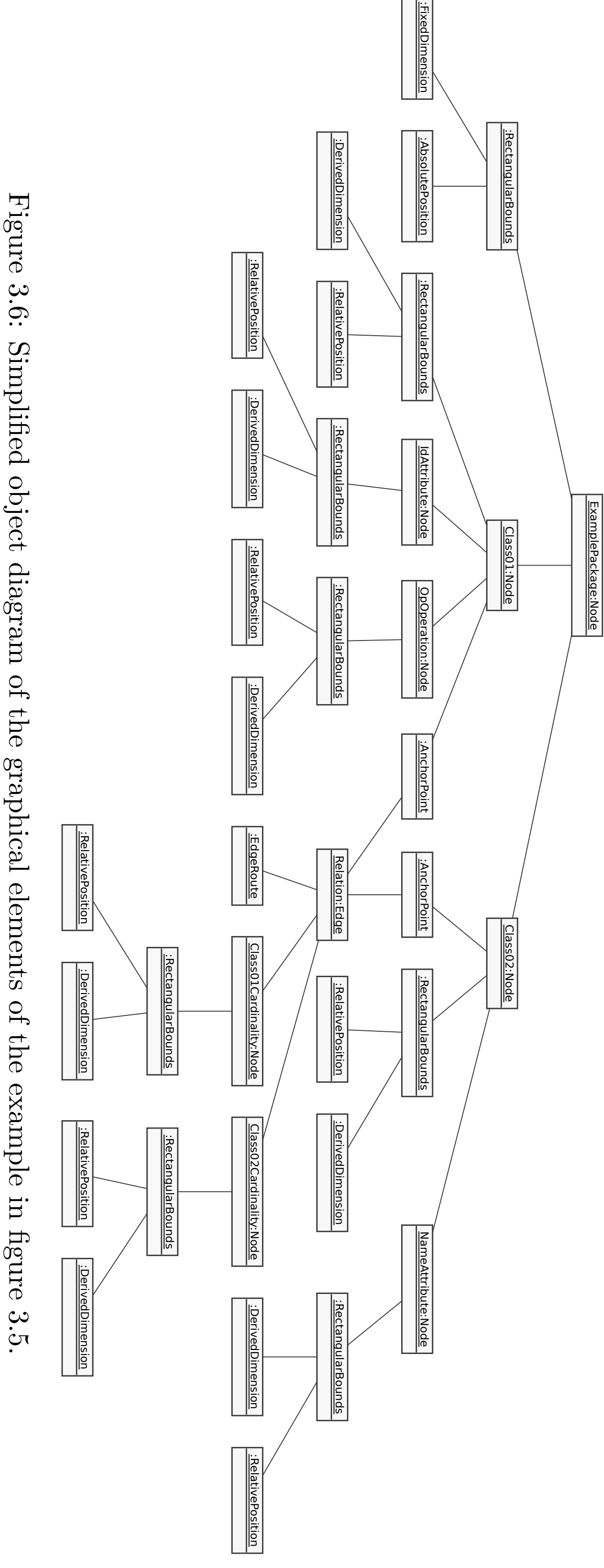


Figure 3.6: Simplified object diagram of the graphical elements of the example in figure 3.5.

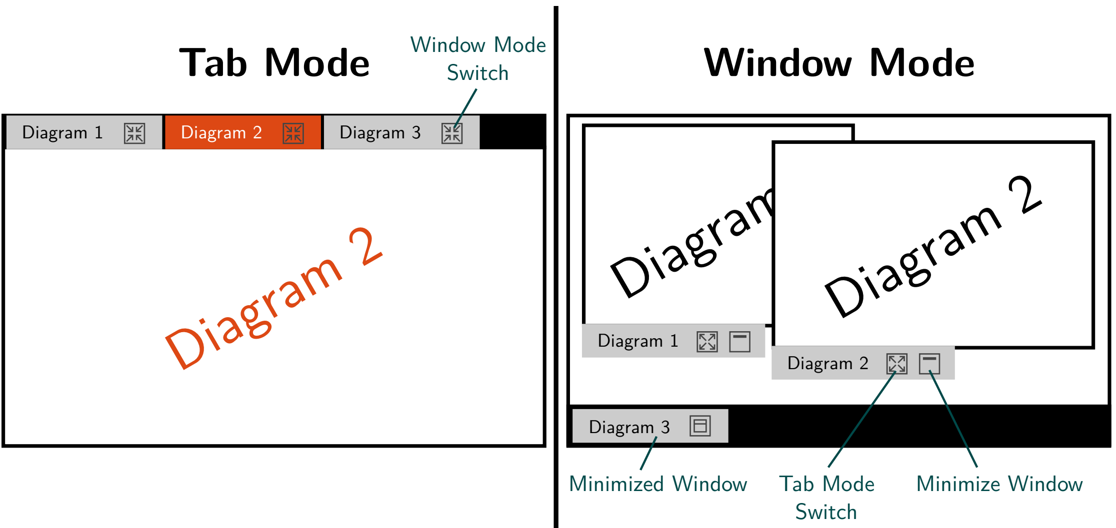


Figure 3.7: Illustration of the unified difference view in tab and window mode.

the layout changes. However, future work may provide layout adjustment algorithms for specific graphical models since interaction cannot solve all problems caused by overlapping elements. Section 3.2.2 discusses some of the biggest problems found while implementing this solution with no layout adjustment in unified graphical models. The unified approach has the advantage that less screen space is used compared to side by side visualizations.

Moreover, models may also be described with more than one diagram which show different aspects of the same elements. So this view also shows all diagrams of the changed artifacts using tabbed browsing or a multiple window strategy, depending on the user preference. In *tab mode*, one active diagram is shown completely in the view, while the other diagrams are hidden. A list of the diagrams at the top can be used to change the currently active diagram. In *window mode*, one or more diagrams are shown in free-floating windows or in an area for minimized windows. Each window's location and dimension can be changed by the user, allowing to place related diagrams next to each other on demand. Additionally, windows can be minimized that show diagrams that are currently not of interest to the user.

3.2.1 Overlays

Graphical elements are annotated by semi transparent overlays. Overlays have been chosen in favor of annotating by coloring the graphical elements directly for two reasons. First, colors might be used by the graphical syntax of the model to encode certain semantics; an element with a special property for example. This might lead to information loss and misconceptions by the user if the colors of a difference annotation is equal or similar to colors in the graphical syntax. Of course, this problem might be partially solved by specifying a different color set based on the graphical syntax. But finding such a color

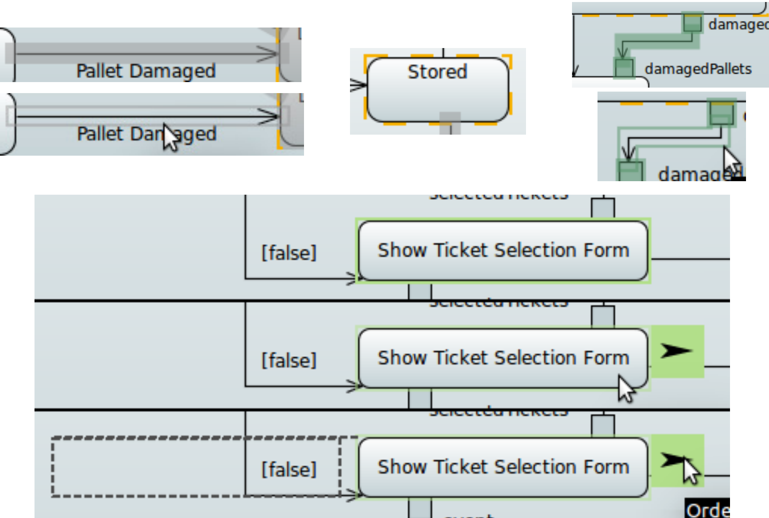


Figure 3.8: Screenshots of various overlays in Mervin.

scheme is not always a simple task and the problem of information loss has not been solved. Moreover, users might be used to certain colors for specific difference types, which may lead to confusion. Second, if no assumptions are made about the graphical syntax, less is known about the graphical elements and how coloring affects the perception of the colored graphical elements. Some elements might be less noticeable than others, or not noticeable at all in the worst case.

Overlays encode multiple types of information of exactly one element: the difference type, layout difference details and the existence of comments. An overlay may contain information of all of these categories and at least information of one these categories. For example, an overlay may represent an addition and the existence of comments if an element is added and a comment references it. Each overlay is a colored semi-transparent rectangle that covers the whole graphical element that represent the changed or annotated model element. In addition, overlays have no interior if they represent only comment annotations and layout differences or a combination of those two. This is due to the assumption that layout differences are less important in most diagrams or graphical models [OWKb]. Comment annotations are intended to remind the user of the existence of comments and less to draw the attention of the user to the element itself. Hence, no interior is drawn for these comment overlays.

The color of the overlay is determined by the information it represents. If the overlay contains atomic difference information, then the atomic difference type it represents is used to determine the color of the overlay. Another different color is used if the overlay represents only a layout difference. The existence of comments is indicated by a dashed outline in another color. In summary, five distinct colors are needed if we consider all difference types mentioned in Section 3.1.

Figure 3.9: Illustration of the overlay types, combinations, indicators and interaction.

Semi-transparency allows the user to see the overlayed element, but does not provide a clear view on the element. Thus, overlays are drawn without interior when the user moves the cursor over the overlay. This allows the user to get a better, unmodified view of the element on demand. So this interaction and the semi-transparency have been done to reduce the coloring problem. If an colored element is overlayed, the color changes due to the color blending with the goal of drawing the attention of the user. Even if this is not the case, the user can check it by moving the cursor over the overlay.

Overlays also have a dependency relation to each other which is derived from the structure of graphical elements they are attached to. More precisely, an overlay depends on another parent overlay if that parent overlay is assigned to an graphical element which is contained in the *parent hierarchy* of the overlay's assigned element. The *parent hierarchy* is the list of the elements when iterating over all parents from a starting element to the root element. Another sort of dependency would be the nodes that are attached to an edge using an anchor, and the anchors itself. So the edge's position and visibility depends on nodes and their anchors. As a consequence, also edge overlays depend on the node's assigned overlays, if they exist.

The dependency relation is important for determining the visibility of overlays to avoid visual clutter, showing dependent layout differences and filtering overlays. Cascading differences, where the addition of one element with multiple nested added elements, are such a source of visual clutter: One overlay is created for the element alongside with another set of overlays for the child elements that very likely overlap the previous overlay. It can assumed that users perceive such additions as a single addition, which has also been implemented as a filter in EMF Compare. Additionally, showing only the parent overlay would avoid visual clutter of overlapping overlays. The dependency relation between overlays can help identify such cases and hide or remove such overlays if necessary. It is also used by the filter implementation of overlay types.

Layout difference details are also shown when the user moves the cursor over the overlay: Location differences are shown with a small arrow that points in the direction that the element has moved. Dimension property changes are shown by glyphs on the corresponding side of the overlay. These glyphs indicate a growth, decrease, fixation or derivation of a dimension property. Fixation is the change from a derived value to a fixed value, derivation is the change from a fixed value to a derived value. Moving the cursor over these glyphs shows a dashed outline of the element's old dimensions at the previous position. This outline may be drawn outside of the viewable area, and the user might have to scroll to view to make the outline visible. To guide the user in the correct direction, a dashed line is drawn from a reference point between the outline and the overlay. A drawback of this method is that if scrolling cannot reveal the outline and the overlay at once, it is also not possible to view the complete layout difference information at once. On the other hand, this method avoids visual clutter and follows the principle of *details on demand* of the visual information seeking mantra [Shn96]. Edge routing differences have no specific indicator, but the previously outline is shown when the cursor moves over the overlay.

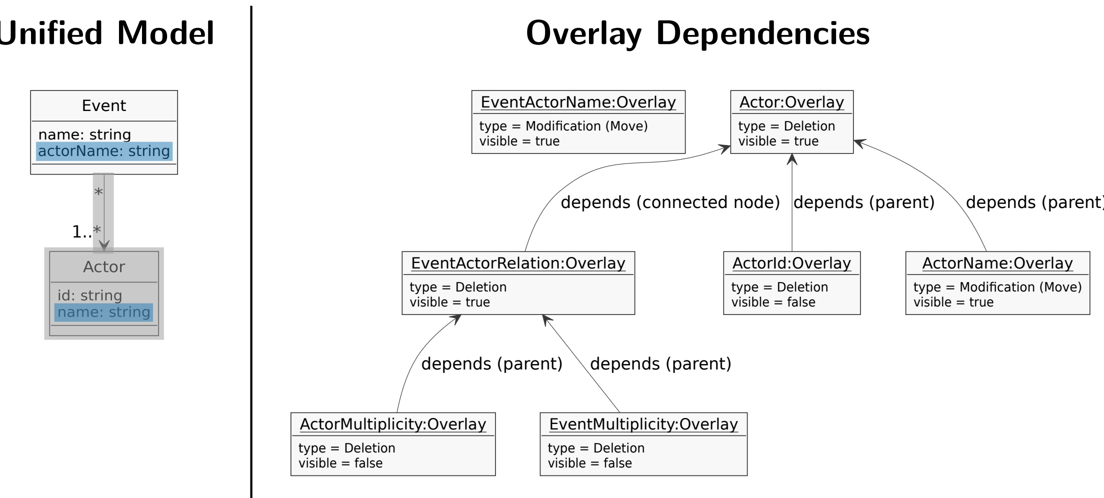


Figure 3.10: Illustration of an example of overlay dependencies.

To avoid confusion, also layout outlines of overlays that depend on that active overlay need to be shown. This may be the case when the user moves the cursor over the glyph or over a deletion overlay if absolute positions are used. The latter is necessary due to the deletion inconsistency problem described in Section 3.2.2. Otherwise node containment can be misinterpreted by the user, as the container node movement is not presented to the user. Nodes that are contained by other nodes may appear as being moved out of another node into the contained node, although the contained node has also been moved with it. The issue was observed after the evaluation and the implementation of the prototype presented in Section 4. As a result, this solution is not implemented in the prototype. However, the issue was not observed during the evaluation and the compared tools did not handle that issue. So it did not affect the results of the evaluation. Moreover, it would also make sense the define model specific dependency relations that are also relevant in this case. This might be of interest for future work as the parent/child/edge relationships are only the most obvious relationships to use for general graphical models.

Users are also able to filter overlays based on the information they represent, including atomic operations, layout differences and comment information. Overlays for the atomic operations *addition* and *change* are simply hidden on demand, whereas the graphical element is not hidden. Overlays that represent *deletions* are hidden alongside with the assigned graphical element, all children of the graphical element, and all depending overlays. So filtering only deletions also affects the display of the unified model, switching from showing the unified model to showing the comparison scope's new model with other overlays. This allows the user to switch between those versions, removing elements that may cause layout problems. As a consequence, overlays are also hidden if they depend on deletion overlays which are not visible. Filters also only hide overlays if they represent no other difference type that is not filtered, as otherwise filtering a single overlay type might result in hiding overlay types which are not filtered. The only exception to that

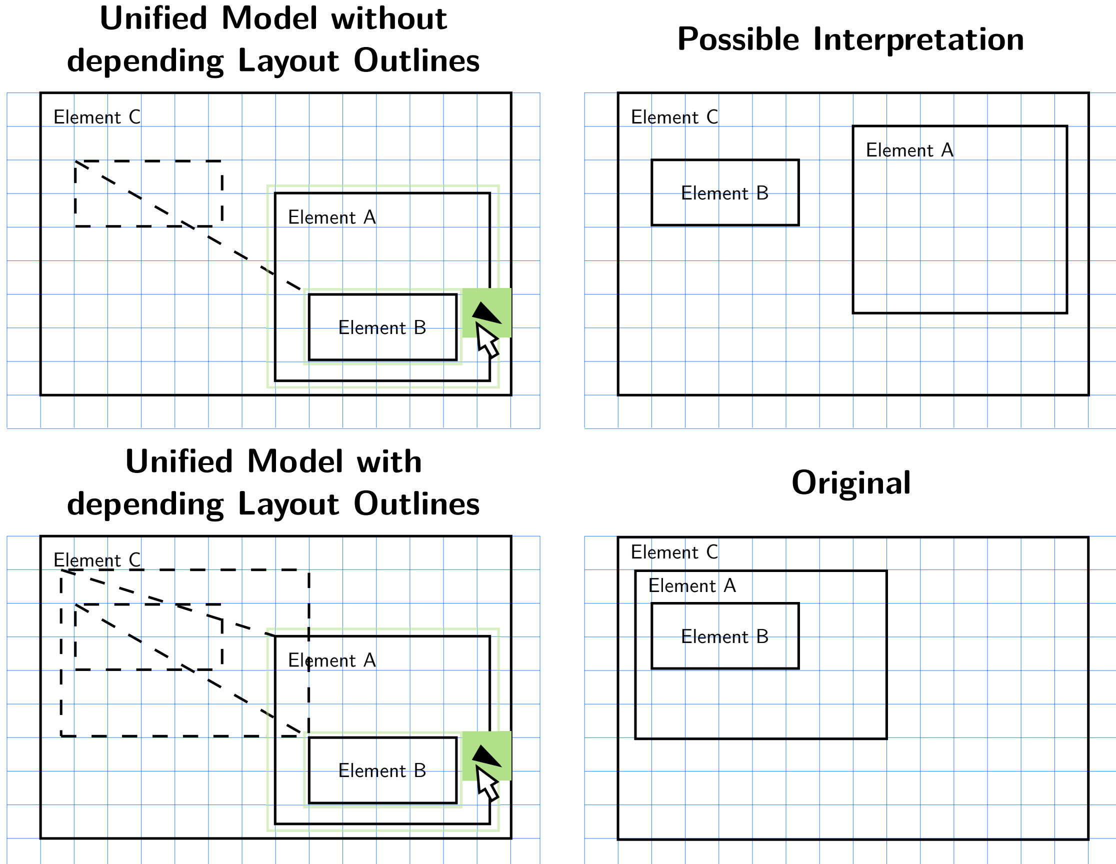


Figure 3.11: Illustration of ambiguous layout outlines.

rule are comments applied to deletions. Otherwise the deleted elements from the old version of the graphical model cannot be completely hidden when comments are applied to deletions. Changing the overlay in the way that the filtered overlay information representation is not rendered on the overlay is also an option which is left over to the actual implementation. But it has to be noted that this may lead to the impression that deleted parts with comments are part of the new version if deletion overlays are filtered.

3.2.2 Deletion Inconsistency Problem

As stated before, adding deleted elements to a graphical model without layout adjustment may cause problems. These are described in this section in more detail with respect to the previously defined structure of graphical elements. Obviously, how layout information has to be interpreted has an impact on which problems may occur. The most simple problems occur when each node and edge uses absolute positions and dimensions with fixed values. Graphical elements may overlap and hide other graphical elements and the user might miss them. Another problem is that the visual containment relationship between deleted

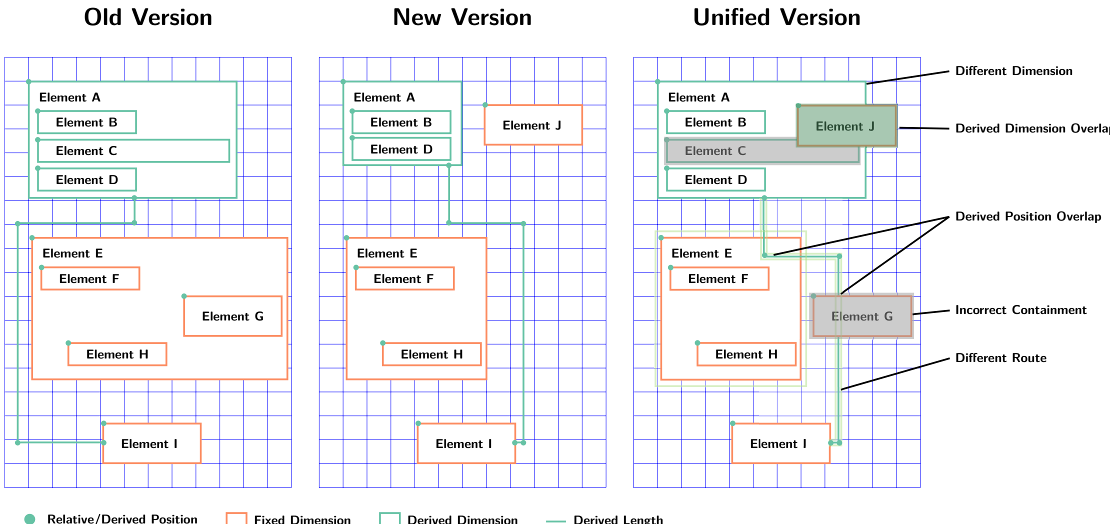


Figure 3.12: Illustration of problematic cases without layout adjustment.

nodes and their parent might get lost if the parent element is moved or resized. In the worst case, users are not aware of this fact and might misinterpret the containment relationship. Even if the user is aware of the problem, the containment relationship remains unclear for deleted elements. On the other hand, this particular problem is mostly solved by the layout difference outline mechanism described in Section 3.2.1. The containment relationship problem also occurs in some cases if only relative positions and derived dimensions are used. For example, if a dimension is derived from a value that causes the node to visually outgrow of its container. Again, also overlaps may occur depending on the reference point and the values to derive the properties from. Relative or derived values also introduce a new problem: The positions and dimensions of the elements differ from their actual values although they have not been changed. Such a difference may also affect one or more derived values, which cause a visual change of an element that does not necessarily need to be changed. Depending on the edge routing approach also the edge routing may change and might be left in an invalid state. Users might take more time to identify the equal elements and are not able to see and judge the layout of the actual diagrams. This is also dangerous in diagrams and graphical models where the layout is used to encode semantic information.

On the other hand, layout adjustment is not an easy task. It requires knowledge about the impact of layout changes on the semantics of a diagram or a graphical model. Such models and diagrams may define layout constrains that must be met in order to be valid. This is especially the case if layout properties like position and dimensions are used to encode information in the model. For example, Unified Modeling Language (UML) sequence diagrams require a different layout adjustment algorithm as simple graph based models like class diagrams. Apart from that, readability is also an aspect that must be considered when calculating a new layout. This includes moving overlapping elements,

rerouting of edges and their related elements and anchor points.

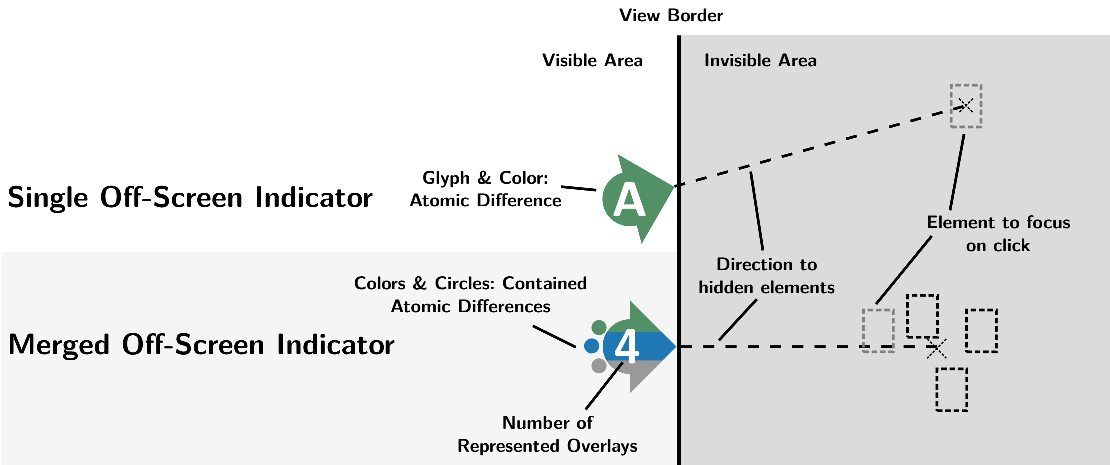
If the layout is defined by the user, it is obviously also valid to check the readability of one version of a diagram or a graphical model during graphical model review. But this is not possible when adding deleted elements as the actual layout is not shown to the reviewer. The user cannot be sure if the layout is the same as in one version, even if no layout adjustment was necessary. So validation of the readability or validation of the layout is another issue that cannot be solved in this case - with or without layout adjustment.

One alternative to solve those problems is to allow the user to interact with the diagram to switch between different versions. This allows the user show a specific version on demand and use his short term memory for comparison. Filtering and hiding deleted elements in the unified view is therefore proposed as a solution for this problem. Another alternative would be not to add deletions to the view, but mark the areas of deleted elements with semi-transparent markers and show the other version in a movable, overlapping frame. However, this "window to the past" approach has been discarded as the connection between identical, but moved elements is not maintained. Further research may provide solutions for this problem but the presented approach uses filtering instead.

3.2.3 Off-Screen Indicators

In general, diagrams and graphical models exist in varying size, so it cannot be assumed that their sizes are fixed or limited, except for technical restrictions. Screen or view space is always limited, and it is obviously even more limited if multiple views share this space. So the view is only able to show a certain area at once and elements outside this area are hidden to the user. This also means that the user must be able to navigate through the graphical elements. For example, scrollbars are a common user interface element for two dimensional representations, that allow to pan the current representation.

Also changed elements are hidden from the user due to the limited visible area, although they might be of interest for the user. Interaction is required to view all changed elements and the user has to memorize where changed elements are located. This is might be cumbersome and the user might miss changed elements. Therefore indicators are shown for off screen elements to guide and help the user memorize changed elements out of the current view. These off-screen indicators are based on the city lights approach [ZMG+03] with additional change information and quick navigation to hidden elements. They are shown as small colored symbols surrounded with a colored circle and a small arrow. Each indicator represents an overlay that is currently not shown on the screen. The symbol and the color indicates the atomic difference type represented by the overlay. If the overlay represents no atomic difference but a layout difference, the symbol and color for layout differences is used. If an overlay represents only a comment, also another symbol and color is shown for the comment overlay. A single click moves the visible area so that the associated overlay is focused and visible in the view. Projection of the centers of the overlays on the borders of the view or screen is used to determine the location of



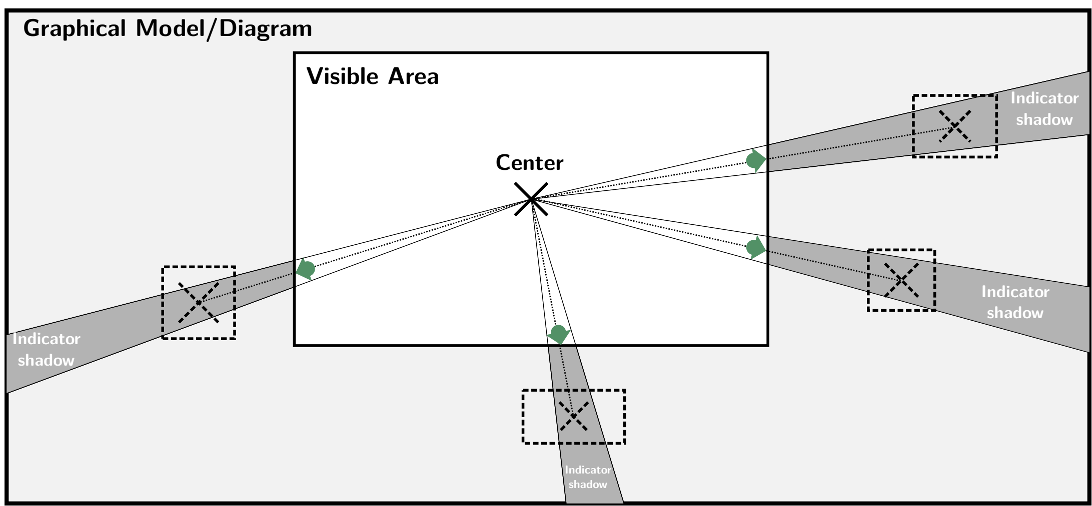


Figure 3.13: Illustration of off-screen indicators.

the indicators. Radial projection has been used in the final implementation in favor of orthographic projection to distribute the indicators more evenly at the borders.

The location is determined in three basic steps: First, a line is constructed that passes the center of the view in the coordinate system of the graphical representation with the center of the hidden overlay. Second, intersect the line with the bounds of the view, which is described by four line segments in the coordinate system of the graphical representation. This results in at least two points and at most in four points, if the line intersects one or two corners. Two points are identical and can be treated as one point if a corner is intersected. Third, take the first point as location with the smallest distance

to the overlay center. As a consequence, indicators move along adjacent edges while panning through the model, making navigation more complex than with orthographic projection [ZMG+03]. To mitigate the problem, each indicator also includes an small arrow that points into the direction of the center of the overlay.

Obviously, all points on a line segment starting from the center to a point at infinity are mapped to the same location. But indicators are not points and cover space. Therefore, a single indicator covers a whole area, and each overlay in that area will have an indicator on the same location. The area is called the *shadow* of the indicator in this thesis. So indicators might overlap due to the projection, which also hides information from the user.

Overlapping indicators are merged to a single special indicator that tries to provide some information of the hidden overlays. The number of overlays that the indicator represents replaces the difference type symbol to give the user an impression how many elements are hidden in the shadow of the indicator. Additionally, the circle and the arrow is striped in the color of the difference types of the represented hidden overlays. Early prototypes showed that the stripes might get hard to distinguish when the number of stripes increases. As a result, small circles representing each included difference type are shown on the opposite side of the arrow. Multiple overlays also imply that no single center exists and pointing the arrow to only one hidden overlay might be misleading. So the direction of the arrow is determined by using the centroid of the centers of all represented overlays instead. Also the navigation needs to be adapted, but in this case the nearest overlay will be focused. The centroid is not used due to the fact that it might be far away from any included overlay. So in some cases the nearest overlay might be out of the visible area when focusing the centroid, which is against the idea of quickly navigating to overlays.

3.3 Property Differences View

The content of every model can be either displayed as list or as a tree in case of hierarchical data. Also properties of the element can be shown as part of the hierarchical data. Models created with EMF for example are stored in a hierarchical manner, the so called *containment tree*, and contain *structural features* which can be shown as properties. This tree or list can be used to show differences in a classical linked side by side comparison, similar to how it is implemented in EMF Compare. Two tree views, where each represents a different version, are shown next to each other and linked with graphical annotations. The property difference view contains such a side by side comparison to assist the reviewer and show differences on properties that are not present in the graphical representation. It usually contains very detailed information of the model and should not contain all differences in the whole model. Only elements related to the current selection in the other views are shown for this reason, following the *details on demand* principle of the visual information seeking mantra [Shn96]. Related elements are the selected element itself and the context of the element, which depends on the actual model.

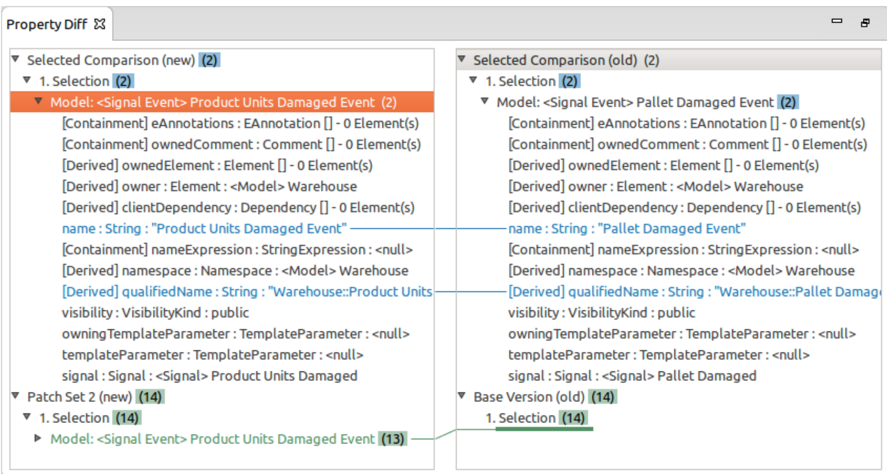


Figure 3.14: Screenshot of the property difference view in Mervin.

The view is not restricted to a single comparison and provides the comparisons in a tree, where the root elements are the comparisons to display. This includes the selected *comparison scope* and the comparisons of the base version with the changed versions. All related elements within a comparison are shown as children. For elements, the properties and the children of the hierarchy are shown as children. Depending on the model, this might results in very broad and deep tree structures. It is expected that users are not always interested in all those items, so each item of the tree is collapsible and collapsed on startup. Expanding and folding elements is synchronized in both tree views, meaning that corresponding items will always have the same state. Added or deleted elements do not have a correspondence in the other tree and are therefore not linked in that way. Some side by side difference views in the presented review tools link the scrollbars of the two sides which can also be done here, but has not been done in the prototype presented in Section 4.

Changed elements are annotated using color and colored lines that indicate the item or position on the other side. The color depends on the difference type. In the case of modified elements, lines simply connect the items that represent modified elements. In the case of added or deleted elements, lines connect the insertion or deletion points with the items that represent the elements. Child elements of added or deleted elements are also connected to the same insertion or deletion points as their parent, if the child element has also been added or deleted. Folding items also hides the annotations which may leaves the impression on the user that there are no differences in the child elements. Therefore each parent element also shows the number of differences per difference type in its whole child tree. Those numbers are colored based on the difference type. Insertion and deletion points are drawn as a thick line which is indented based on the tree level. Equals elements are not colored and no connections are drawn between their items as it is expected that users are more interested in differences. However, a connection is drawn when the user moves the cursor over an item that represents equal elements. This

is especially important in the case if no scroll synchronization is done and the user has therefore less visual cues which elements are equal.

3.4 Comment View

All review tools support to annotate elements of a review by other users and this view is dedicated to such annotations. As the name suggests, the main user annotation in the presented solution is a comment. The simplest comment is a text assigned to a specific version and that comment has been submitted by a person, the comment author. A comment may be a reply to a single comment and may have multiple replies. Parts of the text can be linked to one or more model elements in the version, allowing the comment author to relate the text to model elements. These text links are drawn bold and underlined and the user is able show the linked elements on demand. This is done using highlighting in the other views, either temporarily when moving the cursor over the link text or permanently when clicking on the link text. How this highlighting affects the other views is described in more detail in Section 3.7.

Comments are shown in columns that represent the versions that the comments are assigned to. Which columns are shown depends on the selected *comparison scope*. Versions that are not part of the scope are not shown in detail in this view to save space and allow the display of both columns at once. Nevertheless, the number of comments for each version is shown at the top of the view to make the user aware of comments in other versions. A single comment is rendered slightly smaller than the actual width of the column and aligned either left or right. Comments from the current user are aligned to another side than comments from other users, to allow quick distinction of own and other comments.

While reviewing elements, it might be of use for the reviewer which elements have been commented on in the other version and if the comment has been addressed in the other version. Also, some comments may contain issues that are related and the reviewer need to be aware of them when reviewing. To support that, comments are also grouped, and each group may contain comments of both versions. This way, possible related comments are shown closer to each other, reducing the complexity for the user when tracking related issues. It also shows the history of comments to certain elements in two versions.

The decision how to group comments may depend on the aspects of the actual models, and can be refined by future research. But a general grouping approach can be defined based on the text link targets, as comments targeting the same elements are very likely related. So comments are in the same group if they link at least one same target element. If a comment has replies, all its replies also reside in the same group. All comments that do not link any element are assigned to a single group. As a result, groups contain directly and indirectly related comments, which may include comments which are not of interest to the user. This might be resolved with further information about the model semantics and is therefore of interest for future research. The general grouping approach can be implemented with a two pass algorithm presented in algorithm 3.1.

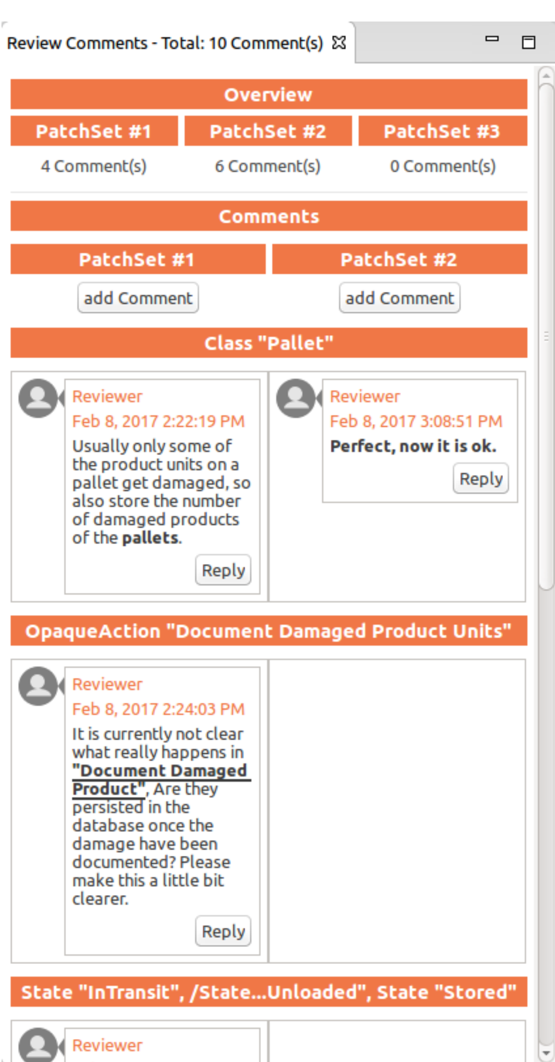


Figure 3.15: Screenshot of the comment view in Mervin.

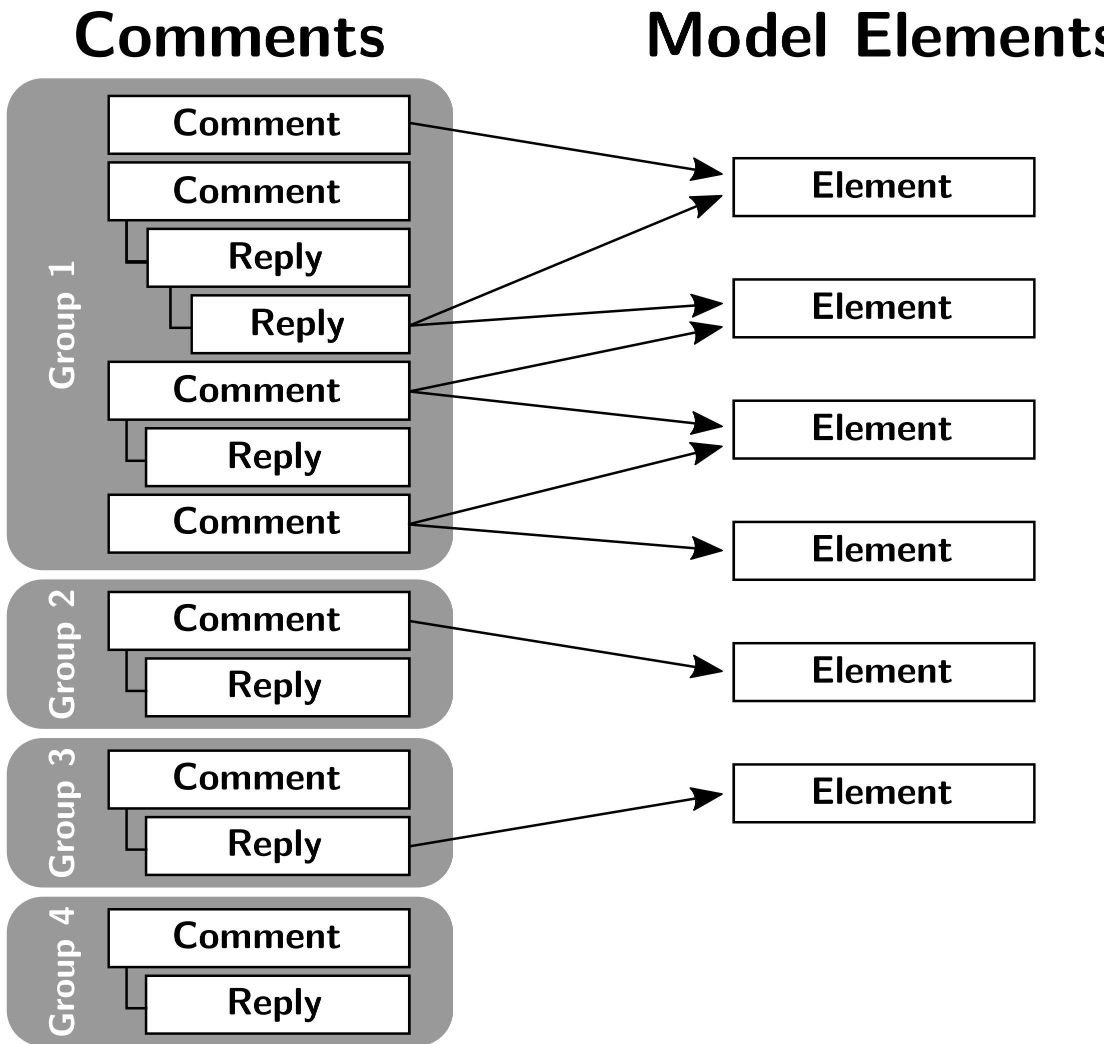


Figure 3.16: Illustration of the general comment grouping approach.

3.5 Version History View

The unified difference view allows to keep track of changes between two versions, but does not provide an easy way to track differences across multiple versions. For example, a specific difference might be the reason why a particular version has been rejected. If such a difference is removed in the next version and reintroduced in a subsequent version, it might be of interest to the reviewer that this change was already present in a previous version. Users can use the comparison scope and track this change by changing the comparison scope, but this is cumbersome and take time due to possible long computation time. Also, all of the other views follow the paradigm of highlighting differences embedded in the model and do not provide a list of all differences. The version history view tries to cover these issues by providing a quick overview of how similar each

Algorithm 3.1: General comment grouping algorithm

```
   Data: a set of all comments
   Result: a set of all groups
   // first pass: create initial group set
 1 foreach comment in comment set do first pass: create initial group set
 2 |  foreach group in group set do
 3 |  |  if comment is part of group then
 4 |  |  |  add comment to group;
 5 |  |  |  break;
 6 |  |  end
 7 |  end
 8 |  create group;
 9 |  add comment to group;
10 |  add group to group set;
11 end
   // second pass: merge groups
12 repeat
13 |  foreach group in group set do
14 |  |  foreach comment in group do
15 |  |  |  foreach other group in group set do
16 |  |  |  |  if other group is the same as current group then break;
17 |  |  |  |  if comment is part of other group then
18 |  |  |  |  |  merge groups;
19 |  |  |  |  |  break;
20 |  |  |  |  end
21 |  |  |  end
22 |  |  end
23 |  end
24 until no groups have been merged;
```

version is in terms of differences and provides a list of all differences.

It contains a combination of a tree view with columns where each item of the tree is collapsible. The first column contains the description of the element represented by the item. This column is followed by a column for each version, containing the similarity metric value for the given element of the item. A similarity value of zero means that this element is not similar to another element in this version, and a value of one means that it is equal to a another element in this version. Values between zero and one give a hint that the element is similar to some extend to another element in the version. The background of the detailed values is colored from dark to bright based on the similarity value to help the user quickly identify high and low similarity values.

The main elements shown in the tree are differences which can be grouped in containers.

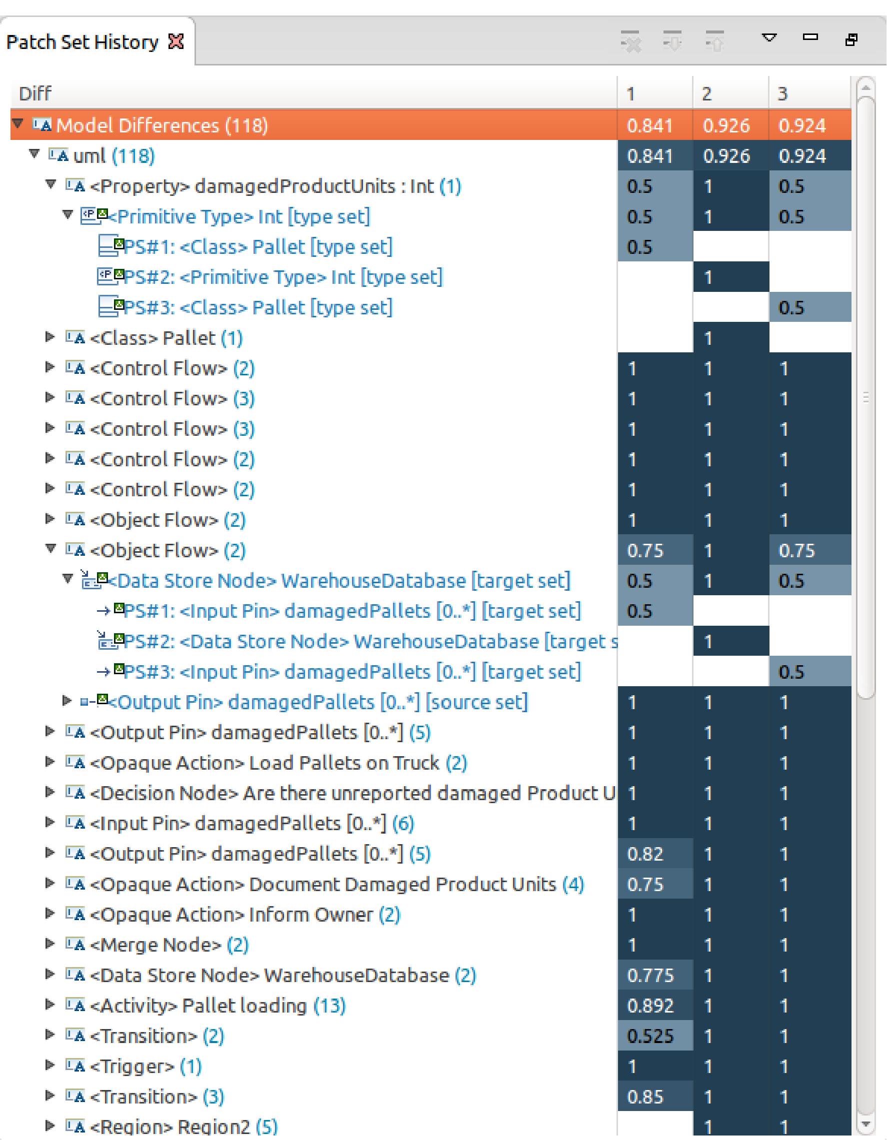


Figure 3.17: Screenshot of the version history view in the prototype with a loaded GMF model.

The label text of these difference items are colored based on the difference type to help the user categorize differences without reading the label text. Similarity values of containers are computed based on their mean non zero child similarity values. Additionally, the child item count is added to the name of the container to give the user a hint how many elements reside in the container. Each difference item also has the corresponding most similar difference item as a child, to give the user feedback which item in another version is similar. How elements are grouped depends on the model structure and the structure that the user is used to. For example, differences of all properties of a UML class element may be grouped by the class. If the model resembles the structure presented in Section 3.2, differences in the child node structure of a parent node can grouped by their parent node. An example grouping method for GMF based models is presented in Chapter 4.

As mentioned at the beginning of this section, this view contains all differences, which also includes differences not present in the versions selected in the current comparison scope. This results in duplicated items for differences which stay the same across all versions, one item for each version. These duplicates can be safely removed from the view, preferably only if the grouping method did not assign them to different containers. It is also recommended to give the user a way to filter the differences to a certain version. For example, the prototype implementation supports filtering differences based on the selected old or new version of the comparison scope.

3.6 Review Explorer View

As mentioned in Section 3.3, models can be shown as a list or as a tree. This representation usually provides a more detailed view on the underlying structure of the model. Also, some tools, like Papyrus or EMF Compare, provide such trees to display models or provide an outline of the model. The idea of the review explorer is to reuse this known structure and annotate it to ease the navigation through the model versions and their differences. This includes users which are used to read this structure as well as users which are not used to read such a structure but use it to locate differences and elements or correlate them with elements of the model.

Therefore, the structure of a model is annotated with difference information of the following set of comparisons: The first comparison is the one that has been selected in the comparison scope. If both versions of the scope are the same, the version structure is shown without differences. The other comparisons are the base version compared to each new version. This allows the user to look up differences compared to the base version without switching the scope. Other comparisons are also possible but have not been used as more space is needed and to avoid misconceptions with the currently selected comparison scope.

Deleted elements are also present at the relative position where they have been removed in the tree. Annotations of differences in the structure are done by coloring affected elements of differences and listing all differences of an particular element in the structure.

The color is applied to the label of the element's item and depends on the difference type. Layout differences are not shown using colors as only their presence can be shown this way and the user has to switch to a view with a graphical representation to properly see the difference in the graphical model. A simple textual description of the layout difference would be an option, but such a description already implicitly exists in the form of atomic differences if layout differences are derived from atomic differences.

Other non-atomic differences that are not part of the structure, or are not possible to be shown as annotations to a certain element, are added to a special difference container item. This item contains all differences as child items that are assigned to the parent item's element. So it provides a way to also show the detailed difference information that has been provided for the parent item's element. Such information may include special derived differences which might not be presentable in one of the other views or using annotations.

Additional columns in the tree view show statistical information about the atomic differences. The first two columns contain stacked bar charts presenting the distributions of each atomic difference type contained in the given item. Users are able to resize these columns to their needs, which allows them to make even small numbers of difference types visible. However, some distributions may contain a very small number of a specific difference type compared to a large number of other difference types. So they are rendered in such small bars that are hardly noticeable, or in bars that cannot be rendered at all. This is might leave the impression to the user that no such difference type exists at all and leads to misconceptions. Hence, two strategies are proposed to reduce this problem. The first one is to show the same distribution bar chart in the second column relative to the total maximum number of contained differences in a tree item. As a result, each bar has more screen space and smaller bars also get more space. Obviously, this won't completely solve the problem. So the second approach is to enhance the stacked bar charts with binary indicators for contained difference types. These are small dots or boxes next to the bar charts that show the presence of a particular difference type. Moving the cursor over the bars reveals the actual number of differences on demand, following the visual information seeking mantra [Shn96]. Also the initial arrangement of the columns follows this mantra. Both of previously columns aim at giving the user an overview about the distribution and the presence of differences in the contained items, where the first has a broader scope than the other. They are also expected to be the most important statistical columns.

All remaining columns show more details as the number of contained differences, the number of contained references to changed elements and other model specific metrics and information. Numeric values of a known range are shown in cells whose background color is interpolated depending on the relative position in the range. Which metrics are useful for a particular model depends on the model and might be part of future research, so it is not covered in the scope of this thesis. However, it is recommended to follow the same arrangement strategy and use background coloring for numeric values within a known range. Moreover, sorting elements based on numeric metric values may also be

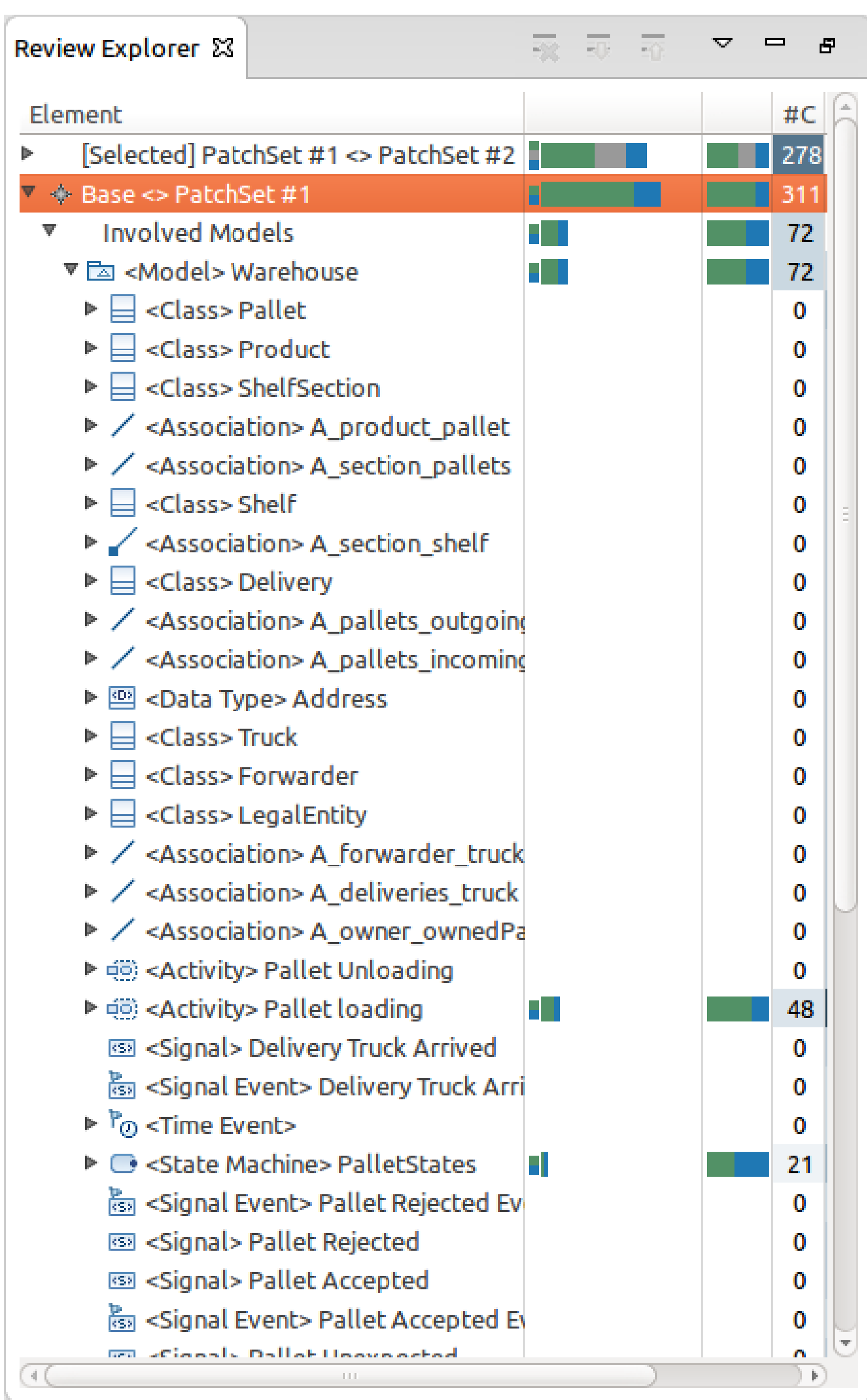


Figure 3.18: Screenshot of the review explorer view in the prototype with a loaded GMF model.

Figure 3.19: Illustration of the difference distribution bar charts in the review explorer view.

enabled for them, like it has been done in the prototype presented in Section 4.

Until now, models and their contained elements have been in the main focus. But those models are also stored and structured in artifacts, for example in multiple files in the file system. Also other artifacts that are provided as a supplement for the information in the model may be part of different versions. Difference visualizations for such non-model artifacts are not part of this thesis, but might be incorporated as a separate linked view that is shown on demand by the user. These artifacts can be shown in the review explorer as it is meant to be the only view that contains also other elements than models. Hence it also is the entrance point for opening incorporated difference views for non-model artifacts.

Depending on the demands of the reviewer and the model, the mentioned structure and artifacts might be of relevance or not. The presented prototype of this solution follows a model centric view, which shows each changed model as a single item in the tree view, despite the fact that they might be distributed among multiple artifacts. However, an additional column shows the artifact that contains the model element of an particular item in the tree, if possible. These models are grouped by artificial items that allow the user distinguish models of different types. For example, the implemented prototype provides two groups: the *involved models* and the *involved diagrams* group, which are based on the special structure of GMF and Papyrus models. The other changed artifacts are shown within their structure in another group that is named *patches*.

This has been done with the assumption that the user reviews the model and its elements, not how the model is distributed among artifacts. However, this can be accomplished by simply restructuring the items of the tree view: Each items resembles the artifact with their original structure and model elements are children of the artifacts that they are contained in. All other columns may also be used with this structure, only the maximum

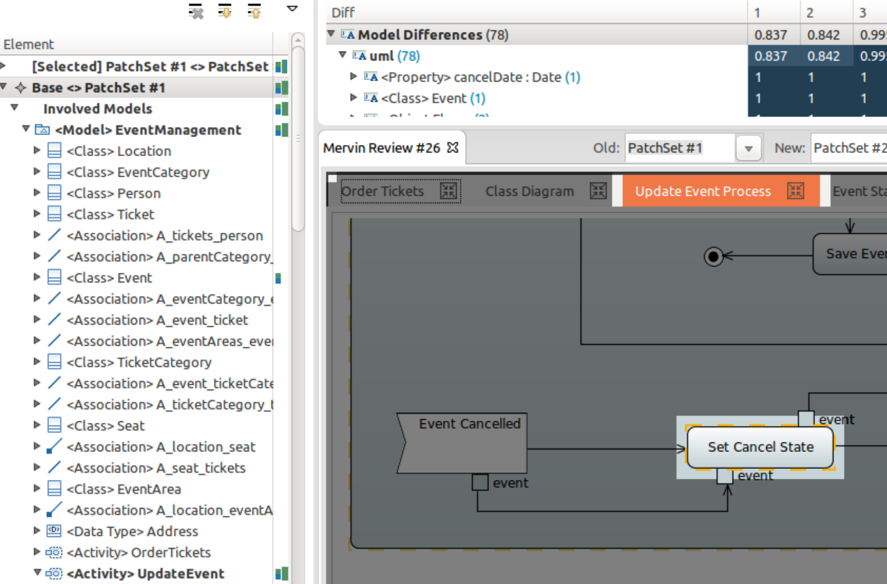


Figure 3.20: Screenshot of highlights in Mervin.

number of total differences might be different.

3.7 View Coordination And Highlighting

All presented views are coordinated and linked with each other using two approaches: The first approach is that they view different aspects of the same review. They use shared data like the comparison scope and the current selection to filter their contents based on the user input, like in the *property difference view*. This type of coordination has already been presented in the previous sections for the particular views. But another important linking technique is proposed in this solution, which is user defined element highlighting. Most of the views show different aspects of the same elements, which possibly makes it hard for the user to identify the same or related elements in other views. Highlighting is meant to help the user to quickly navigate to these elements and show them in a different view, if possible. It is assumed that the *review explorer view*, the *version history view*, and the *unified difference view* are the views where the user has to often switch between them and that the *comments view* is often used to trigger highlights. Hence, highlighting is expected to be mandatory for these views and has been implemented for this views in the prototype.

Highlighting can be temporary or permanent, with the option to clear the highlighted element on demand. How elements are highlighted in a view depends on the view, but the highlighting has been done with the following consideration: A highlighting approach should draw the attention to certain elements, but should should not hide other, potentially important elements completely. Therefore, two different types of highlighting

are proposed: Tree based views use bold text for items that represent elements that are highlighted or contain elements that are highlighted. This approach cannot be applied to the unified difference view as for the same reasons why overlays have been used for annotating changed graphical elements. As a result, a semi-transparent gray layer is drawn on top of the contents in the unified difference view except for areas that contain highlighted elements. These elements might not be visible if the view is in *tab mode*, so the tab that contains the element is highlighted instead.

Element highlight requests are triggered by the user by selection, cursor movement or context menu. The former two trigger the highlight of the currently selected or pointed element and can result in permanent or temporary highlights. A selection might also contain multiple elements if the view supports it. All elements of the selection are highlighted in this case. The context menu usually triggers permanent highlight of derived elements of the current selection. Which elements can be derived depends on the actual model and might be subject to future research. For example, the prototype implementation presented in Section 4 allows the highlight of referenced elements in other models. Previous highlights are cleared and replaced with the new ones if the user triggers a new highlight request for other elements.

Users are able to quickly navigate through all permanently highlighted elements using the tool bar in the tree based views. It provides two buttons that allow to quickly focus the next or previous highlighted item in the tree, relative to the currently selected element. Quick navigation is not supported in the unified difference view as no natural order of highlighted elements exists. An option would be to use the typical reading direction in text documents, but this does not necessarily be the case for all different types of diagrams and graphical models. Another option is to define the direction based in the model semantics, but this depends on the actual model and is therefore subject to future work.

3.8 Required Input Data

Based on the previously defined views, the proposed solution requires a specific input data set and other information about the models and diagrams. The input data set presented below is described in relation to the review artifacts presented in Section 2.2.

Version, diagram, model data The integral part of the proposed solution are diagrams and graphical models in the corresponding versions. This includes the version that contains all original *base artifacts* that have been changed, as well as all versions of the *changed artifacts*. These artifacts within a version may contain one, more or parts of models, graphical models and diagrams, that together form a set of changed models, graphical models and diagrams. Other artifacts may also be present and might be shown in the proposed solution, but require additional visualization techniques that are out of the scope of this thesis.

Comparison information To display differences and allow interaction of the displayed differences, the comparison information is needed. This may be provided by an algorithm that provides the information on the fly or as a precomputed set of differences. Difference information of all possible combinations of comparisons are needed in the latter case. Atomic differences such as *additions*, *deletions* and optionally *modifications* must be contained in these comparisons. Besides that, also matching information of equal elements must be provided.

Diagram and model information The proposed solution draws the graphical models or diagrams and annotates them. So it requires information on how to draw them, or a framework that draws them and provides the layout information for a particular element. Such a framework needs to allow drawing the overlays on top of the graphical representation. Another option would be to provide already rendered versions of the graphical representation and information on how to locate and extract graphical elements on demand.

Review data Also data about the review must be provided. This includes comments containing links to model elements as well as the current and other user data. It is also necessary to provide information on how the version is represented.

CHAPTER 4

Implementation

A prototype has been implemented to prove the feasibility of the previously described solution. It is called Mervin and published under the terms of the Eclipse Public Licence (EPL) [Ecl17b] on GitHub. The complete code is available at `https://github.com/theArchonius/mervin`. Although its matured state, it is considered as an experimental application which is not actively maintained and not meant to be used in production environments. Also, some of the described features could not be implemented completely in some cases due to problems with the used frameworks. These cases are described in more details in the subsequent sections. Mervin is based on various plugins of the Eclipse Platform. The most important ones are EMF, Graphical Editing Framework (GEF), GMF, EMF Compare, and Papyrus.

Papyrus is a set of plugins that allows to create UML Models based on GMF. GMF is a framework that allows to define and create graphical models based on GEF and EMF. GEF is another framework that allows to define and create editors based on graphical elements. EMF is the base framework for models used by various eclipse based applications. EMF Compare is a framework that is able to compare EMF based models with some extensions that also allow to compare Papyrus models.

Apart from that, Mervin also needs review data from a review tool. Gerrit [Goo17b] is a popular review tool and used by most eclipse projects for that purpose, so it has been chosen to be supported with Mervin. Not all features and steps of the whole review process of Gerrit are currently supported. Only the parts that have been described in the previously presented solution is covered. Therefore, Mervin uses and manipulates the review data provided from Gerrit but does not trigger Gerrit actions for a particular change. So version comparison and comments are supported, but the process of accepting and merging a change is not supported and has to be done with Gerrit.

The repository contains the needed Eclipse plugins based on Maven and the necessary eclipse target platform. It is possible to build the plugins using Maven, but the exported

product from maven does not work at the time of this writing. However, it is possible to start the product from the product editor within eclipse when the provided target platform is correctly configured. Note that the target platform is based on dependencies not under the control of the author of this thesis and therefore their availability cannot be guaranteed. If the builds fails for that reason, contact the author of this thesis who will try to resolve the problems if possible.

During the development of Mervin several issues raised with GMF. Some of them could be resolved with workarounds, but some could not be solved without fixing some bugs in GMF. They have been fixed in a fork of the offical GMF repository as only slow development has been observed on the official repository during development of Mervin. The fork is available at `https://github.com/theArchonius/gmf-runtime` and the applied changes can be found in the branches *gmf-runtime-fixes-latest-release* and *gmf-runtime-fixes*. Builds based on the release branch can be obtained from `https://bintray.com/thearchonius/gmf-runtime-fork/gmf-runtime`. As the prototype itself, these builds are considered as experimental and not meant to be used in a production environment.

The property differences view and the review explorer view described in Section 3.3 and Section 3.6 use tree views with entries derived from the differences reported by EMF Compare. No specialized or complex logic related to differences or review data has been implemented for that purpose, except for the atomic difference mapping described in Section 4.5. Also no major challenges arose during their implementation, so the detailed implementation for these views are not described in this chapter. Interested readers are therefore encouraged to take a look at the source code provided in the Git repository mentioned above. The same applies to the comment view, except for storing and loading of comments as well as how the comment model looks like. These topics are discussed in detail in Section 4.4 and in Section 4.3.

4.1 Papyrus and GMF Overview

As mentioned before, Papyrus is based on GMF, which is itself based on GEF. GEF provides user interface elements that can be used to create graphical editors based on the commonly used Model View Controller (MVC) pattern. This is done by extending base controllers, so called *EditParts*, that provide the logic to update a set of view components and the model according to the developers needs. Each graphical editor may have multiple controllers which are part of a tree. A controller in that tree is created and assigned to a single model element, and specifies which model elements are derived from the assigned model element to create child controllers for them. Child controllers are created by a separate provider which takes the child model element as an argument. So the controller creation is based on the model and changes to the model may cause the creation or destruction of new and existing controllers. Changes to the models are done using *commands* to support transactions and undo or redo of actions. User interface editing tools may request these commands from controllers and their *edit*

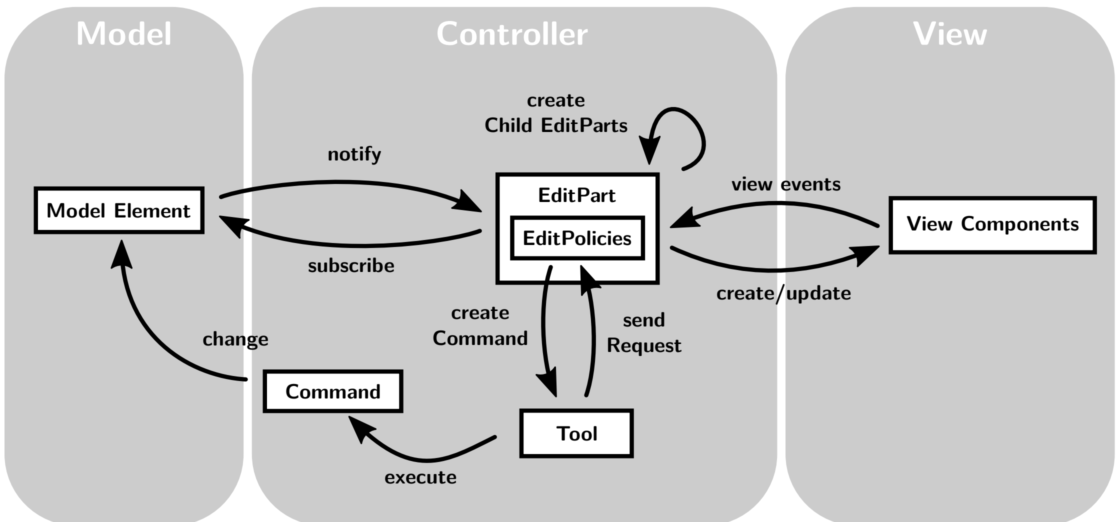


Figure 4.1: Simplified GEF architecture overview

policies before execution. However, editing is not allowed during model review, so editing support has been disabled for the unified view in Mervin. View components are created by the controllers and usually also reflect the hierarchy of the controllers, except the view components that are assigned to special layers. Such a layer is a simple view component in the tree that acts as a container and is itself contained in another container that stacks the layers. Each view component specifies a layout manager that defines how child view components are arranged. However, the order in which child components are drawn is defined by the view component itself.

GMF extends GEF by extending controllers and the command infrastructure of GEF for EMF models. But the framework does not provide the model to controller mapping and controller creation mechanisms, as they have to be defined for a certain EMF model. However, GMF does not use the EMF model directly as input model. It uses the so called notation model, which contains references to the actual EMF model, the so called semantic model. This separation does not necessarily mean that they are also stored in separate resources. The notation model contains the description of the graphical elements and the properties that are not stored in the semantic model, or cannot be derived from the semantic model. It has to be noted the controllers may choose to ignore the values defined in both models.

A *view* is the main model element in the notation model and usually references exactly one model element in the semantic model. However, not every model element in the semantic model has to be assigned to a single view. In fact, it may be referenced by any number of views or even by not a single view in the notation model. *Views* may also contain child views.

Three different types of views exist: *Nodes*, *edges* and *diagrams*. Nodes may have a

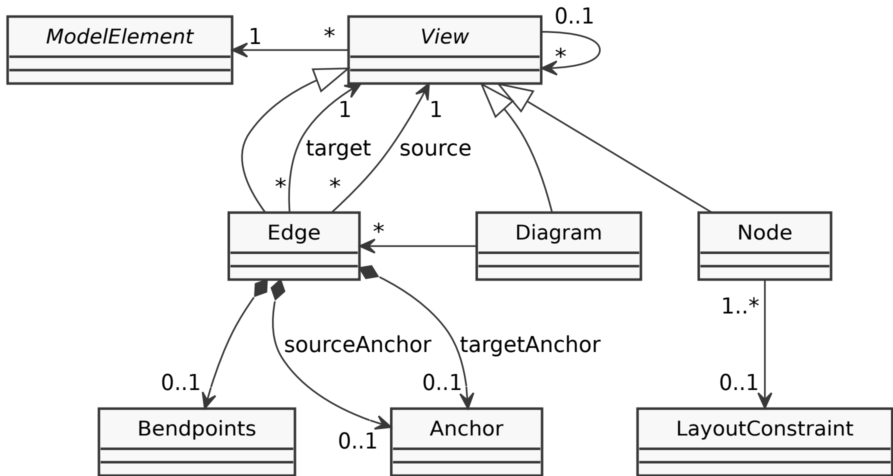


Figure 4.2: Simplified GMF notation model

layout constraint assigned which describes data that is used to layout the node. GMF provides a set of layout constraints but any implementation that uses GMF may choose to add other layout constraints. Moreover, the interpretation and use of these constraints depends on the actual controller implementation. Edges always have exactly a source and target view, where each have an assigned anchor. *Bendpoints* are stored to describe the route for an edge. Again, the actual routing behavior is defined by the controller and is not stored in the notation model. Diagrams are usually the root element of every notation model and should contain references to all edges within the diagram. The model does not constrain the use of edges as child views in a node, but GMF and GEF do not support this case without a manual adaption of the provided controllers. A special connection layer is used for edges by default, which is maintained by the controller of the diagram.

Papyrus provides the actual implementations of GMF classes for Papyrus UML models. It uses most of the GMF infrastructure but also adds customized implementations of controllers and view components. The notation model has not been modified in the presented structure, except that more than one diagram may be stored in the same notation model file and each references a part of the Papyrus's semantic model. Papyrus models are usually stored within three different file types: the *di* file, the notation model files, and the semantic model files. The new *di* file contains internal layout information of the user's papyrus editor, like open diagrams.

4.2 EMF Compare Overview

The semantic as well as the notation models of Papyrus are EMF models. Mervin uses therefore EMF Compare [Ecl17c, Ecl17d] to compare EMF models contained in change versions. More specifically, Mervin loads the models of change versions in so called

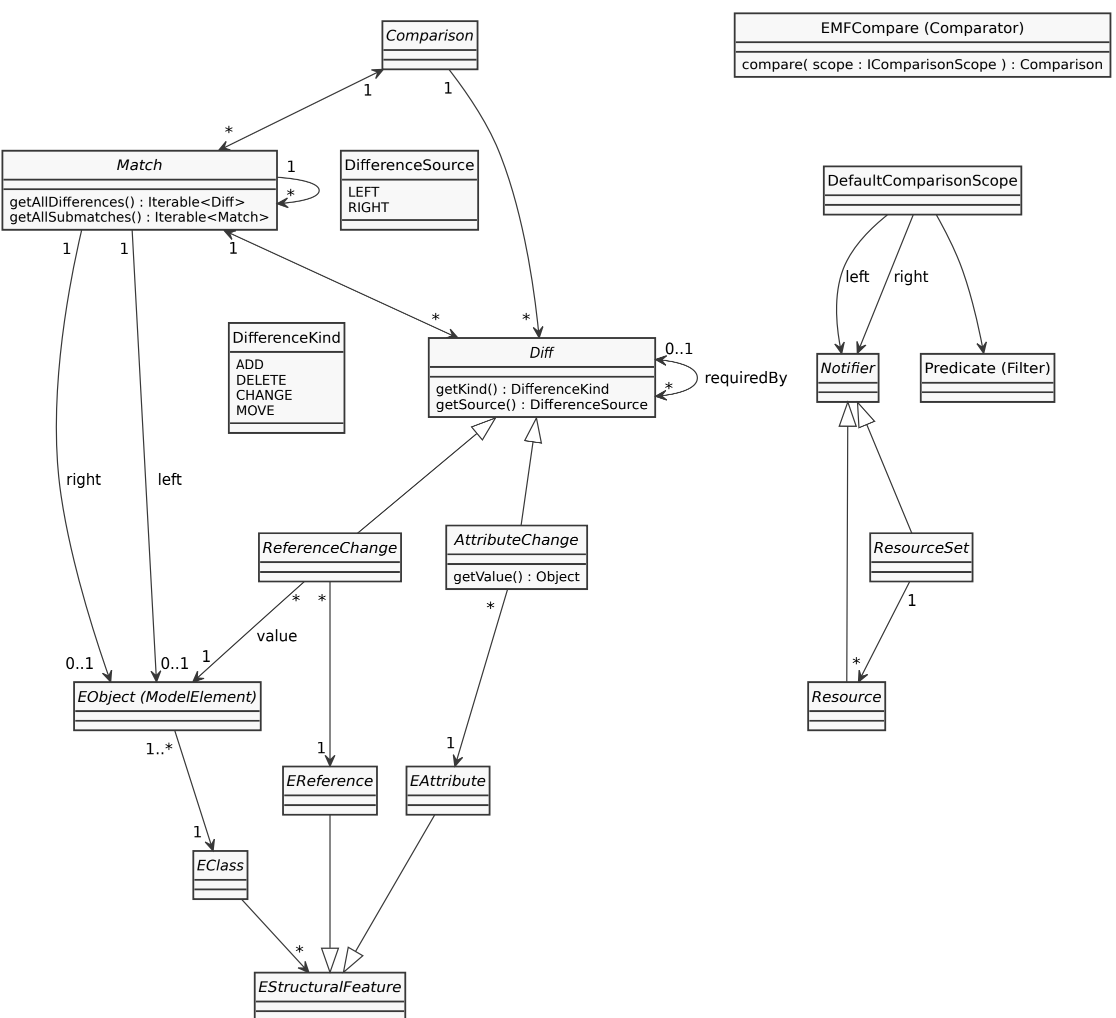


Figure 4.3: Excerpt of the architecture of EMF Compare and its dependencies used by Mervin.

resource sets that will be compared. This is done by passing a *comparison scope* to a *comparator* which returns a *comparison* model. Additionally, resource filters are applied to the comparison scope to restrict the comparison to a particular model type when necessary. EMF Compare supports two way and three way comparisons, but Mervin uses currently only two way comparisons. A comparison model is created after comparing two models with EMF Compare, which provides various informations about the differences of the given models. Mervin mainly makes only use of two types of artifacts in that model: The *matches* and the *diffs*.

A *match* represents two elements that EMF Compare considers to be the same model elements. Each is contained in a different version of a model, although the elements might have changed between versions. They are called the left and the right model elements.

It is also possible that an element has no match in the other model version. A match is also created in this case, but without a reference to a model element in the other version. A match may also contain matches for contained child elements, contained in so called *structural features*. So the matches of a comparison are a forest of trees that show matching information between two model versions within the comparison scope. These matches are created using a *matcher*. Each match lists the differences of its child features in a *diff* element for each difference.

A diff element may represent different subclasses of differences and provide additional informations for the types they represent. Mervin makes only use of the diff subclasses *AttributeChange* and *ReferenceChange*, which provide difference information for references and attributes in a model. Every diff also has a common set of attributes and the most important ones are the difference kind and the source. A difference kind encodes the difference as an atomic operation that have been applied to the old version of an element to transform it to the new version. The source defines which version is the old version as EMF Compare uses *left* and *right* to denote the versions. Another important aspect is the dependency information stored in each diff element. A diff element is dependent on another diff element if it cannot be applied to the old version without applying the other element. Mervin uses this information to determine elements to include in the unified difference view and to organize diff elements in the review history view.

EMF Compare also supports merging and conflict detection. Therefore, it provides additional information and elements in the comparison model that are necessary for these operations. While this is the case, none of these features are required to implement the proposed visualization and have not been used by Mervin except for the elements mentioned above.

4.3 Internal Review Model

Mervin uses an internal review model that is based on the findings in Section 2.2. It is shown as a diagram in figure 4.4. Note that it uses different names for several model elements as it was developed for use with Gerrit. The *ModelReview* element represents a change, and a version of the change is represented by a *PatchSet*. There are two main elements in the model: the *ModelReview* element and the *DifferenceOverlay*.

A *ModelReview* element stores all information of the loaded change as well as the cached comparison of the currently selected comparison scope and the active overlay filters. Besides that, another important property is held by the *ModelReview* element: the unified model map. This map contains the mapping of GMF views in the original notation models to the copies that are used in the unified difference view. A single view in one of the original models may have multiple copies, and a copy may only be copied from one view in one of the original models. This map is used in combination with the unified difference view and therefore more details can be found in Section 4.7.

As mentioned before, *PatchSets* represent versions in a change and contains references to

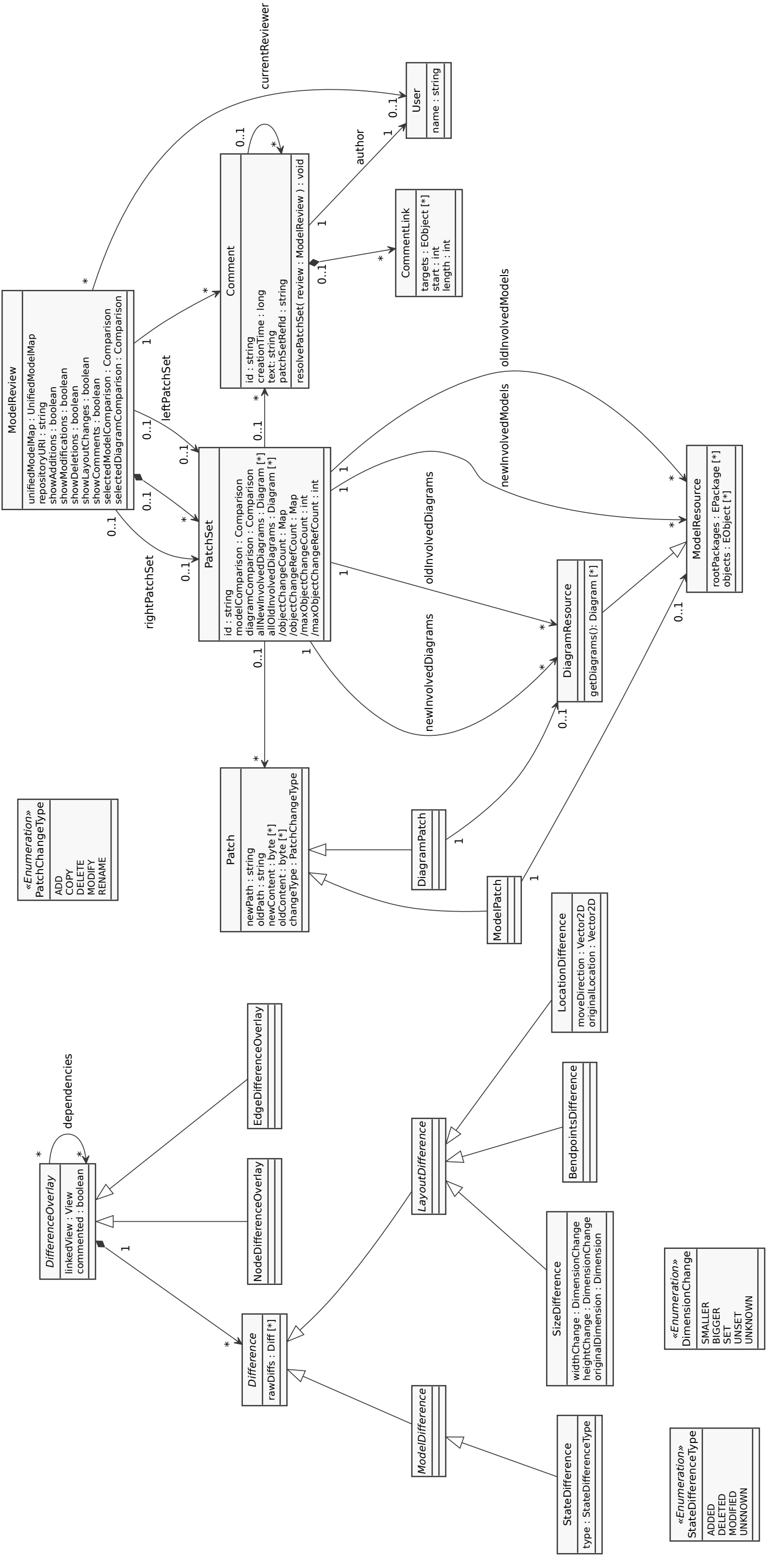


Figure 4.4: Mervin review model.

derived statistical data, patches and their referenced model resources. A *Patch* represents a changed file and stores the information provided by the VCS. Such a patch can be a model patch containing the changed semantic model versions or a diagram patch containing the changed notation model versions. If none of this applies, a patch simply represents an unknown file type. *ModelResources* and *DiagramResources* represent the particular model or diagram version affected by a patch. As such a patch cannot exist without an old or new version, at least one resource of them must be specified. The resource also contains the loaded model elements, the so called *EObjects*.

These model elements can be referenced by links in comments, where each link is assigned to a sequence of characters in a comment. A comment is always assigned to at least one *PatchSet* and is always written by an author. The patch set reference property is meant to store the id of the referenced patch set when storing the comments, as the patch set element is not persisted. Comments also may have one or more replies to them, such that a user can directly address issues mentioned in a comment. A *ModelReview* element also contains a reference to all comments that are referenced by its containing patch sets. It also contains comments that could not be assigned to a patch set while loading the comment. This might happen if, for whatever reason, a patch set is not loaded and the stored patch set id references this missing patch set.

Difference overlays represent the overlays that may be drawn in the unified difference view. They have no relation to the model review for several reasons: First, they depend on the currently selected comparison and therefore there should be an relation between the comparison and the overlays. Second, the only option to solve that would be to extend the comparison class by EMF Compare, as it is defined in EMF Compare. Although this would be possible, additional customization of the EMF Compare match engines and their registry would be needed to actually use this extended comparison. But this option has been dropped to focus on the development of the proposed solution as the relation was not needed during development. Two different overlay types exist, one for nodes and one for edges. They exist mainly to distinguish the creation of the appropriate controllers in GMF, but do not provide more information than the type of the overlay. Also, edge difference overlays only support differences related to edges, which are currently only instances of the class *BendpointDifference*.

An overlay shows one or more *Differences* which describe the various differences described in Section 3.1. Each difference element may be derived from one or more diff elements from EMF Compare and the references to those diffs are also stored in the difference. They are stored for further reference as only those values are derived that are actually needed for the proposed overlays in the unified difference view. Differences are further divided in layout differences and semantic model differences. Layout differences contain a more detailed description for the particular layout difference type based on the low level diffs from EMF Compare. The only exception is the *BendpointDifference*, as a detailed description of the changes to an edge could not be extracted alone from *GMF* notation models. Further details about this issue is described in Section 4.6. *StateDifferences* describe differences on the semantic model expressed in atomic operations on the old

semantic model. This difference type is also the only type used for semantic model differences.

Most of the review model is loaded when loading the data from Gerrit and a few parts like comments and the current comparison are set and changed during the review. Comments and their authors are persisted, all other model elements are derived when loading a change. A more detailed overview of the loading process is given in the next section.

4.4 Gerrit Integration

Gerrit uses Git extensively to store review data and manage the review process. A version of a change is represented by a commit, a so called *patch set*, which is based on a base commit in the main development branch. Each commit message contains a unique change id which is used to identify version commits for a given change. By default, contributors amend the version commit if they have to change it, which results in a new commit which is also based on the same base commit. Comments, as well as other actions are handled by the web interface or the public API of the Gerrit server. Each version commit is stored in the remote Git repository with a special Git reference that points at the commit. This reference contains also the internal numeric change id, so it is always possible to find all references for given change. Additionally, Git references can be transferred between repositories like every branch in Git, so it is possible to access the version commit data completely using only Git commands.

These properties of Gerrit have been used by Mervin to load the change data from Gerrit. The only exception are Gerrit comment data, which are not modified by Mervin. Mervin comments are stored apart from normal Gerrit comments due to the fact that they contain also links to model elements which are not supported by the Gerrit web interface. All remaining actions can be done using the web interface or the API of the server.

A user has to clone the remote Git repository from Gerrit to load a change for the repository. This does not necessarily include the Git references that contain the change data. Mervin uses the remote repository to fetch these references on demand when loading the change. Algorithm 4.1 gives an overview of the process that initializes the review model defined in Section 4.3 based on the change data. First the low level information of changed files is loaded into patches which are stored in patch sets that represent a version in the change. The type of the patch model is determined from the file type of the file represented by this patch. Binary content data for the old and the new version of the file as well as other common patch metadata are assigned afterwards. More details about the patch loading process are shown in algorithm 4.2. Patches that contain model data are then loaded using EMF using so called resource sets. EMF supports lazy loading and distribution of model data across multiple files and uses therefore these resource sets to find and load references. Each version is loaded with its own resource set, but they use a single resource set cache to avoid duplicated loading of resources. This cache is also needed later to resolve references to model elements in the comments, as they

reference elements in multiple versions. The cache contains all resource sets that have been created during the loading of the review model.

Once all patches and model data are loaded, two comparisons are done using EMF Compare and stored in the review model. One that compares the notation model and one that compares the semantic model. This is done by comparing the resource sets of the old and the new version with a filter that distinguishes between the two model resource types. After loading all patch sets, all patch sets are sorted by their internal Gerrit id to allow faster lookup of the patch sets. This id is a number starting at one in each change and is incremented by one for each new version. All comments are loaded in the last step.

Algorithm 4.1: Overview of the review (change) model loading process.

```
   Data: internal change id, Git repository
   Result: the loaded review model instance
 1 fetch all Git references of the change;
 2 create review model and set metadata;
 3 initialize model resource set cache;
 4 foreach Git reference in Git repository references do
 5 |   if is reference to change version commit then
 6 |   |   create patch set model and set metadata;
 7 |   |   load patches;
 8 |   |   extract model resources with model resource sets cache;
 9 |   |   update model resource set cache with new resources;
   |   |   // update cached comparisons
10 |   |   create & cache semantic model comparison with base commit;
11 |   |   create & cache notation model comparison with base commit;
12 |   end
13 end
14 sort patch sets;
15 load comments with resource cache;
```

As mentioned before, model data loading is handled mainly by EMF with cached resource sets. However, this was not the only adaption that had to be done. References in the model files used to load model elements are stored with Uniform Resource Identifiers (URIs) that contain the file path, but do not contain the version that they are contained in. This is perfectly fine for loading a checked out version in a Git repository, but leads to ambiguity when loading models without checking them out each time a model element needs to loaded. Although the latter is possible, it is very cumbersome and the local Git repository's working directory may not be clean, as it may be altered by the user before the checkout. Another problem is that the resource set caching gets complicated as the URI alone cannot be used to decide if a resource of a particular version has been cached or not. Therefore, two adaptations have been done: First, a URI converter is installed for

Algorithm 4.2: Overview of the patch loading process.

Data: patch set, patch set Git reference
Result: the patch set with loaded patches

```
1  extract list of changed files between version commit and base commit foreach
   changed file in the list do
2      if is semantic model file then
3          create semantic model patch model and set metadata;
4      else if is notation model file then
5          create notation model patch model and set metadata;
6      else
7          create patch model and set metadata;
8      end
9      set change type of patch;
10     set old & new path of patch;
11     load and apply new & old binary data;
12     add patch to patch set;
13 end
14 return the patch set;
```

each resource set that contains a commit hash referencing the commit that contains the version. This information is used to create a custom Git URI that contains the commit hash and the URI from the model file. EMF calls this converter when loading resources, so every URI is enriched with the version information. Second, a custom URI handler is added to the list of the default URI handlers. EMF uses these handler when it actually loads model elements for a resource. This custom handler contains a reference to the local repository and supports loading the enriched URIs from the local repository.

With these modifications, each model file is loaded by creating a resource with the corresponding custom Git URI. Once this is done, also the involved models and diagrams are extracted and added to the patch set model. A more detailed view on the process is given in algorithm 4.3.

Comments are part of the review model and as the review model is also an EMF model, also EMF is used to store and load the comments. However, not the whole review model is persisted as it is mainly derived from the change data. Therefore, comments are stored with the referenced users and weak references to patch sets. These weak references are simple strings that contain the corresponding patch set id. They are replaced with the actual reference during loading and are created while storing comments. Comments with a reference to a patch set that does not exist are ignored, although this case should never happen in practice. No code review tool allowed deletion of versions, and comments cannot be assigned to patch sets that do not exist.

Comments are stored and loaded from a custom Git reference, namely `refs/mervin/-comments/changeid`, where `changeid` is replaced with the internal Gerrit id of the

Algorithm 4.3: Overview of the model resource extraction process.

```
   Data: patch set, patch set Git reference, cached model resource sets
   Result: the resource sets for model and diagram patches
 1 create resource set with cache lookup for old patches;
 2 apply file to Git URI converter to old patches resource set;
 3 apply Git URI handler to URI converter;
 4 create resource set with cache lookup for new patches;
 5 apply Git URI handler to URI converter;
 6 apply file to Git URI converter to new patches resource set;
 7 foreach patch in the patch set do
 8     if is semantic or notation model patch then
 9         create custom Git URI for new version;
10         if patch has not been deleted then
11             create resource in new patches resource set with new version Git URI;
12             load new version of patch in resource;
13             update patch set model list;
14         end
15         create custom Git URI for old version;
16         if patch has not been added then
17             create resource in old patches resource set with old version Git URI;
18             load old version patch in resource;
19             update patch set model list;
20         end
21     end
22 end
23 return old & new patches resource set;
```

Change. Due to the fact that not all elements of the internal Mervin model are stored in the comment resource, all comment elements are copied with their references and then stored or assigned to the review model. Again, the custom URI handler and converter are used to load the resources for the different versions. Also the resource set cache is used again, and serves the purpose of resolving the model element references of already loaded resources. This way no element is loaded twice from the same resource although they are in different resource sets. The process of loading the comments is shown in algorithm 4.4 and the process of storing them is shown in algorithm 4.5.

4.5 EMF Compare Difference Mapping

As mentioned before, low level differences detected by EMF Compare have been used to derive layout and model differences in the review model. No further mechanisms have been implemented to compute differences that cannot be derived form the notation model

Algorithm 4.4: Overview of the comment loading process.

```
   Data: review model, Git repository, cached model resource sets
   Result: the review model
 1 if comment ref for the review exists then
 2 |   create resource set with cache lookup;
 3 |   apply file to Git URI converter resource set;
 4 |   apply Git URI handler to URI converter;
 5 |   create resource with fixed comment URI;
 6 |   load the comment resource;
 7 |   copy comments & users in review model;
 8 |   resolve patch set reference for all comments;
 9 end
10 return the review model;
```

Algorithm 4.5: Overview of the comment data extraction process while saving comments.

```
   Data: review model, comment resource
   Result: comment resource
 1 create list of elements to copy;
 2 add all comments to the list;
 3 add all comment authors to the list;
 4 add all patch sets to the list;
 5 copy all elements and maintain references between copies;
 6 foreach comment in copied list do
   |   // patch sets are not stored in the comment resource,
   |      therefore replace it with an numeric id
 7 |   set numeric patch set id as reference;
 8 |   remove patch set reference;
 9 end
10 add copied comments in resource;
11 add copied users in resource;
12 return the comment resource;
```

and from the semantic model. Such mechanisms require in depth knowledge about the implementation of all controllers in Papyrus which is out of the scope of this thesis. This section gives more insight in how atomic differences from EMF Compare are translated into difference model elements used by Mervin.

The most simple mapping has been used for atomic changes on the semantic model as they are mapped directly from the corresponding diff element. Two options exist to determine the diff element to use: First, the diff element for the containing reference of the element in the semantic model can be used. However, a changed element in the semantic model does not necessarily mean that also a corresponding view element in the notation model exists. As a consequence, a difference is created for an element that is not shown in a graphical representation of the model. Hence also no overlay can be shown and this difference can only be shown in one of the other views which also show non-graphical model elements. Difference information of the EMF Compare diff elements can be transformed on the fly and used in the latter case, which is also how Mervin handles them in the other views.

The second option is to use the diff of the containing reference of the view element in the notation model that represents an element in the semantic model. Mervin uses this option to determine the diff element for semantic model changes which are visible in the graphical representation. This is a valid option as other differences cannot be linked to graphical elements in the diagram using only the notation model. Further information about the mapping is needed in order to fix this problem. As a consequence, such differences are not supported by Mervin, but future implementations may provide a mechanism to provide the missing mapping information.

Apart from that, EMF Compare's *DifferenceKinds* can be simply mapped the Mervin's *StateDifferenceTypes*. *ADD* and *DELETE* are mapped to *ADDED* and *DELETED*. *MOVE* and *CHANGE* are mapped both to *MODIFIED*, as Mervin does not distinguish between those two types.

Layout constraints and edge routes are harder to map, due to the fact that they cannot be obtained without further knowledge on the actual implementation of the controllers and the layout managers. The issues are explained in more detail in Section 4.6. As a consequence, only detailed layout differences are created for node elements which are placed using rectangular layout constraints and *XYLayout*. Differences of the location are used to determine the direction of the move in a normalized vector. Width and height differences are mapped according to the change of their values. An increase is mapped to *BIGGER*, a decrease is mapped to *SMALLER*. If the former value was below zero, and therefore dynamic, it is mapped to *SET*. In the other case it is mapped to *UNSET*. Fortunately, most of the nodes are placed in that way in Papyrus models. Other layout constraint differences are currently ignored but may be also considered in future implementations. Bend point differences do not contain additional information and will be created if differences are detected on the bend point reference of the edge.

4.6 Obtaining Layout Information

GMF stores layout constraints for nodes in the notation model. These constraints are passed to the controller which defines how they are translated to layout constraints that are used to compute the actual layout of the graphical element. Again, additional information from the implementation is necessary to interpret these correctly. Fortunately, most graphical elements in GMF are placed using a so called *XYLayout* that uses a layout description based on rectangles. Such a constraint represents the location relative to the upper left point of the containing view and a width and a height. Hence, Mervin is only able to visualize relative positions which are translated to absolute positions at runtime based on the merged containing node. In theory, this results in an incorrect position if a node is moved between two different containers and the move is detected by the difference detection algorithm. However, this behavior has been accepted for Mervin as GMF usually destroys the view and creates a new one when moving elements between containers, which was not detected as a move during the development.

Values below zero for the width and the height are interpreted as the width or height that is necessary to contain all child view elements. So the width and the height can be set to a fixed value or may be dynamically determined on runtime. Again, such dynamic values cannot be determined without further knowledge about the actual size of the children. The width and height of the figure in the unified view will be used to show the outline of the old version in this case. Other layout information is not interpreted by Mervin, but all layout constraints are kept and used during the merging process in the unified difference view.

Edges are routed using bend points and anchors which are stored in the notation model. Like layout constraints for nodes, these bend points and anchors are passed to the controller and then transformed to routing information that is used to obtain the actual route. Although mostly only one or two routing mechanisms are used, several other problems prohibited a meaningful interpretation of the bend points in the unified view. Due to the fact that edges are often used to connect nodes in a diagram, their positioning is also tied to the position and size of their connected nodes. As mentioned before, such nodes are relatively or dynamically positioned. This is no problem for determining the layout of the edge in the new version of the model. But determining the layout of the old version is problematic if the layout of the connected nodes have been changed. Although it is possible to determine the old layout of some of the connected nodes as described before, some problems exist with this approach: First, an edge may connect a node whose layout could be determined with a node whose layout cannot be determined. Second, bend points and anchor positions are also often based on dynamic values or relative values. Reusing them without the exact old layout may result in completely different edges, and should not be presented to the user if possible. These problems leave just a small subset of edges whose old route can be accurately displayed. So the decision was made not view the old edge routes, as the reasons stated before are not aware to the user. Therefore the selective display of the old routes of edges might confuse more than it supports the user.

In summary, dynamic and relative values are the main problems that need to be addressed before layout information can be completely obtained for GMF models. One solution would be to load the old version separately and perform the layout computation for it. Another solution would be to provide an API for graphical models developers who may provide layout resolvers for graphical elements. These resolvers may then be used by the visualization implementation to obtain the exact location and size for all graphical elements. However, the first one has not been implemented as it requires major adaptations to the used frameworks. Note that in both solutions, global properties like screen size and font size may also affect the final layout of the graphical models, which also may change during runtime. The second solution has not been implemented as this would require the specification of layout resolvers for all supported UML elements of Papyrus, which would go beyond the scope of this thesis. Mervin provides therefore only partial support for layout difference visualization as described in this and the previous sections.

4.7 Unified View Implementation

Mervin creates the unified graphical model every time the comparison scope changes and the comparison has been applied to the review model. This is primarily done by copying notation model elements in the unified model. GMF is used by the unified view to create the graphical model to display, as this allows to use the same controllers as Papyrus. Some adaptations have been made to make this work as GMF does not provide a way to include other model elements out of the box and Papyrus also added support for custom styles.

The model review element of the presented internal model in Section 4.3 is the root semantic model element for the view. Papyrus diagrams are shown as tabs in the unified view and so each diagram in the new version of the Papyrus model is a child of the root element. However, no copies of these diagram views are used to represent them, as additional features, like the overlays and the off-screen indicators, must be supported by the controller and the view component that draws the diagram. Moreover, the Papyrus controllers of these diagram views create additional layers that are not needed and would interfere with the layers needed by Mervin's view components. So a new Mervin diagram view is created as a child of the root view for each diagram in the new version of the Papyrus model. All remaining child views of the new Papyrus model are copied and assigned as child views to corresponding Mervin diagrams. A *model hint* is also added to each view such that GMF is able to resolve the same controllers and view components as Papyrus. This also applies to view copies of the old Papyrus model which is described later in this section. Also note that copying in this section means that the element as well as its contained elements are copied.

Note that diagrams are not part of this view if they have been deleted in the new version of the model. This is the intended behavior as otherwise two cases would be shown in the same manner, which makes the visualization ambiguous. In the first case, the diagram

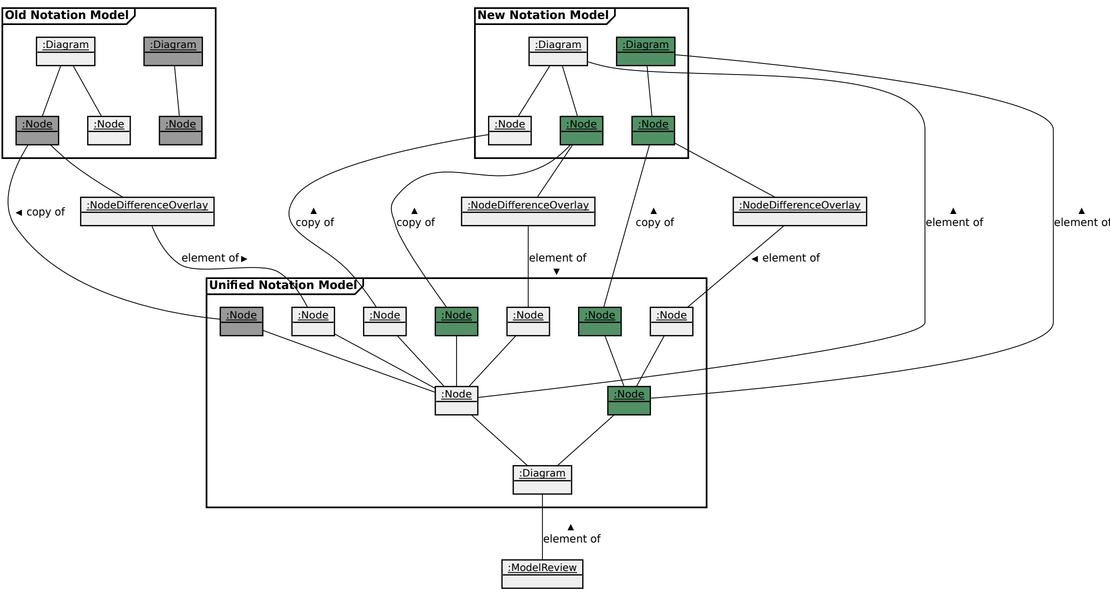


Figure 4.5: Simplified object diagram of an example unified notation model in Mervin and its corresponding Papyrus notation models.

itself might have been deleted and therefore also all of its contents. In the second case, only the content is deleted, but the diagram itself remains. As a result, the first case is handled by showing no diagram in the unified view, and the second case is handled by showing the diagram with its old content with deletion overlays. Other views, like the review explorer and the property difference view can be used to show the deleted diagram in the first case.

Algorithm 4.6 and algorithm 4.7 give an overview of how the unified notation model creation process is implemented. Mervin always clears all views except the root model from the unified notation model before creation of the unified model to simplify the process at the cost of performance. However, further research may provide an approach to reuse notation model elements to speed up the process.

So far only the new version of the Papyrus model is incorporated in the unified notation model. As the unified model also contains elements of the old version of the Papyrus model, deleted views also need to be added. This is done with the help of a so called unified model map. Its main purpose is to link the original notation model elements to their copies and vice versa. One might think that only one copy per notation model element exists in the unified model, but there is a problem during the unification process that is resolved in such a way that this assumption is not true any more.

For example, there might be an element A that has a reference to a single element B, and the notation model does not allow to add a second element C to this reference. So the reference of element A is a *mono-reference*. In the case that element B is deleted and replaced by element C in the new version, the *mono-reference* becomes an *multi-reference*

Algorithm 4.6: Update process of the unified notation model

Data: rootDiagramView (root notation model element), modelReview (root semantic model element)
Result: unified notation model

```
1  clear unified model map;
2  clear unified view model in root diagram;
3  diagrams = empty list;
4  if new version is the base version then
5  |  diagrams = diagrams of base version;
6  else
7  |  diagrams = diagrams of new patch set;
8  end
9  foreach diagram in diagrams do
10 |  mervinDiagram = create mervin diagram for diagram;
11 |  add mervinDiagram to rootDiagramView;
12 |  createUnifiedDiagram(mervinDiagram, diagram, modelReview);
13 end
14 return rootDiagramView;
```

in the unified model. So a reference constraint of the model has changed. This is exactly the case when the source or the target of edges are changed to a different node. GMF and the Papyrus controllers are most certainly not designed to handle such edges correctly, and changing the meta model of the notation model during runtime is also not possible. There have been only two options to address this problem: The first is to create a unified notation meta model based on the notation meta model, where each mono-reference is replaced by a multi-reference. As a consequence, also each controller of papyrus has to be adapted to handle these changes to the meta model. The other option is to copy an element if such a constraint violation is found. Mervin introduces therefore so called *pseudo copies* for simplicity reasons. It also creates such pseudo copies only for references of a deleted element that is copied into the unified diagram. This was done as this problem was a only a corner case and this approach was sufficient to create a unified Papyrus model. A holistic approach would be to create these copies for all reference constraint violations and handle the case of circular references. As a result, each element in the unified model map may have one or more copies assigned, where only one of the copies for an element is not a pseudo copy.

Doing so raises the question how to display such pseudo copies. In the case of views which are placed using the common *XYLayout*, no problem exists. They would be drawn exactly the same way above or below the other copies. Therefore, users are not aware that a second copy is drawn, as Papyrus uses no semitransparent view components to draw elements. It has to be noticed that these pseudo copies also occur when an edge is rerouted from a deleted node to a new or existing node. Other layouts require additional

work: The corresponding view components have to be moved to an additional layer which places them at the same position as the non-pseudo copy. Again, this would have required to replace all controllers provided by Papyrus. As mentioned before, this would have required huge additional work that would be out of the scope of this thesis. So Mervin takes a different approach: It creates a deletion overlay for pseudo copies as they represent the old reference. This introduces ambiguous display of pseudo copies for view components with layouts that result in exact overlaps, but users are able to use deletion overlay filters to identify these pseudo copies. Another feasible solution would be to create a new overlay type for these elements, but this has been discarded for simplicity reasons.

Algorithm 4.7: Unified model creation for a specific Papyrus diagram

Data: mervinDiagram (unified notation model), diagram (original notation model), modelReview (root semantic model element)

Result: unified notation model for the given diagram

```
1  root elements = collect all edges and visible children of diagram;
2  root elements copies = copy root elements and update references between copies;
3  update unified model map based on the copies, including child copies;
   // merge deleted elements
4  diagramMatch = get match for diagram in current modelReview comparison scope;
5  view copy map = create new map;
6  foreach difference in all differences (including child matches) in diagramMatch do
7  |  if not already copied then
8  |  |  copy and add deleted view in view copy map;
9  |  end
10 end
11 update references in copied views;
   // apply unified model
12 add copies to mervinDiagram;
13 add overlays to mervinDiagram;
14 return rootDiagramView;
```

With the unified model map, two additional steps have to be done to create the unified model: First, the deleted old notation model elements must be copied, as well as all pseudo copies. Second, these copies must be incorporated in the copied notation model of the new version which includes updating references of the old element copies to the new element copies where applicable. In GMF, every diagram has root views and the root views of both diagram versions can be easily combined into a single set of unified root views. Mervin assumes that the order of the elements is not relevant, which is true for Papyrus diagrams. If this is not true, additional information is needed for this merging process. This merging can be done right after the view has been copied, as the container of the copies is always known before copying the view. Conversely, other references cannot be updated during the copying process, as the corresponding copies do

not necessarily exist at this time. They are updated in a second step, after all copies for the unified models have been created. These copies are stored in the unified model map, as well as in a *copy map* that is maintained by a *copier*, an utility provided by EMF to copy model elements while maintaining the references among the copies in the map.

Mervin iterates over all entries in the copy map of the deletions and pseudo copies to update all remaining references of the old element copies to the new element copies. This is described in more detail in algorithms 4.8, 4.9, 4.10. First the non-containment references are replaced by the new element copies in the unified model. The unified model map is used to find the correct copies in this step. As mentioned before, multiple copies may exist for a single original element. In a multi-valued ordered reference all copies of a value replace that original value and all subsequent values are moved back in the list of referenced values. If the value is not a known original value in the unified model map the matched value in the current comparison scope is used, as new elements might be part of the unified model. Otherwise, no replacement is done to keep references between copied elements, which also includes pseudo copies. Afterwards, the containment reference of each element is updated. Unifying means that the old element copies must be inserted into the corresponding containers in the unified model, instead of just replacing values in references. Mervin does that by placing them right after the first unchanged previous sibling, but does not maintain the order of deleted elements. This is a bug of the prototype, and a corrected algorithm that uses the merged structure of EMF Compare is shown in algorithm 4.10.

Algorithm 4.8: Update references of copies created while merging deleted views

Data: copy map
Result: updated references of merged deleted views

```
1 foreach entry in copy map do
2 |   foreach non containment reference of copy class do
3 |   |   update reference;
4 |   end
5 |   update containment reference;
6 end
```

As mentioned before, only the deleted view elements of the old model are copied and merged into the unified model. However, this copying process is more complex as it seems at first glance. One problem is for example that differences depend on each other and must be applied before applying the difference during merging. The container is an example for such a dependency. A child element cannot be merged without also merging its container if it is not part of the model. Although the container is guaranteed to be copied when using the presented algorithms and the implemented comparison scope, it has been decided to design this method also for the case a single view element must be copied without using the other algorithms. Therefore, all required view differences reported by EMF Compare will be copied using the same technique before copying the

Algorithm 4.9: Update reference of a copy created while merging deleted views

```
   Data: reference, original, copy, unified model map, diagram comparison
   Result: updated references of merged deleted views
 1 if is multi-value reference then
 2     foreach original value in reference of original do
 3         if unified model map contains original value then
 4             replace original value with copies of original value in copy reference;
 5         else
               // fallback: use copies of matched value, otherwise
                  keep value as is
 6             if copies for matched new original value exist then
 7                 replace original value with copies of matched new original value in
                    copy reference;
 8             end
 9         end
10     end
11 else
12     value = value of reference in copy;
13     if copy value is not a copy then
14         if value has copies then
15             if value has multiple copies then
                   // should never happen due to pseudo copies
16                 return ILLEGAL STATE ERROR;
17             end
18             replace value with value copy;
19         else
               // fallback: use copies of matched value
20             if matched new value has copies then
21                 if matched new value has multiple copies then
                       // should never happen due to pseudo copies
22                     return ILLEGAL STATE ERROR;
23                 end
24                 replace value with value copy;
25             end
26         end
27     end
28 end
```

Algorithm 4.10: Update container of a copy created while merging deleted views

```
  Data: original element, copy element, unified model map
  Result: updated container reference of copy element
1 if container of copy already set then
2 |   return;
3 end
4 new container = matched new container of original element;
5 if matched new container has multiple copies then
  |   // should never happen due to pseudo copies
6 |   return ILLEGAL STATE ERROR;
7 end
8 if containment reference is mono-valued then
9 |   apply copy element to new container copy;
10 else
11 |   get child matches of original container match;
12 |   find previous match whose element copy is present in new container copy;
13 |   if such a match exists then
14 |   |   add copy element after the element copy in the new container copy;
15 |   else
16 |   |   add copy element at the beginning in the new container copy;
17 |   end
18 end
```

element, if this has not been done before. It also allows to check if the container is present in the resulting copied model, and skip copying a view element that cannot be assigned to a container in the unified model. Once the copy has been made, the unified model map is updated with the copy.

Pseudo copies are created before the view copy is made. Each reference in the element is checked whether its opposite reference is mono-valued to check if a pseudo copy is necessary. If that is true, and the container of the element that holds the opposite reference exists, then a pseudo copy is created and the unified model map is updated accordingly.

The resulting unified model is then applied to the corresponding Mervin diagram view, as well as the overlays for this diagram are created and applied. Mervin iterates through the whole unified view tree and adds overlays if necessary. An overlay is added in various cases: if a state difference or layout difference is found as described in Section 4.5, if a comment is assigned to that view, or if it is a pseudo copy. This operation is done using the original views which are retrieved using the unified model map.

Comments may be assigned to elements of the new or old models that are shown in the diagram, whether it be an element from the notation model or an element from the semantic model. Therefore, comment references are checked in both versions for

Algorithm 4.11: Copying deleted views

```
   Data: view element difference, copy map, root elements copies
   Result: updated copy map
 1 foreach required difference in view element difference do
 2 |  if not already copied then
 3 |  |  copy and add deleted view in copied map;
 4 |  end
 5 end
 6 if container copy of view element to copy exists then
 7 |  copy view and add to copy map;
 8 |  copy non-unifiable references;
 9 |  if container is a diagram then
10 |  |  add copy to root element copies;
11 |  end
12 end
13 update unified model map;
14 return copy map;
```

Algorithm 4.12: Copy non-unifiable references (pseudo copies) of copies created while merging deleted views

```
   Data: element, root elements copies
 1 foreach reference in element do
 2 |  if referenced element references this element and containing reference is
 |     mono-valued then
 3 |  |  if container copy of referenced element to copy exists then
 4 |  |  |  copy referenced element and add to copy map;
 5 |  |  |  mark as pseudo copy in unified model map;
 6 |  |  |  if container is a diagram then
 7 |  |  |  |  add copy to root element copies;
 8 |  |  |  end
 9 |  |  end
10 |  end
11 end
```

that particular view and its referenced semantic model element. Both view versions are obtained from the corresponding match from the comparison. As multiple views may reference the same semantic element, only the topmost view in the *parent hierarchy* that references that element is considered in this check. The same logic is also applied to the state differences to avoid visual clutter as described in Section 3.2. As mentioned before, a deletion overlay is also created for pseudo copies. The overlay that has been created for the closest view in the *parent hierarchy* is assigned as a dependency for each overlay that is created. Hence, an overlay is independent if no overlay exists for a view in the *parent hierarchy*.

An index of views to overlays is also maintained during creation of the overlays as it is needed to increase the performance lookup in the final step: Mervin updates the overlay dependencies between edge overlays and their views connected node's overlays, if they exist, by iterating over the created node overlays. The index is used to collect the overlays of the connected edges and set the node overlay as dependency of the edge overlays. This dependency relation is used when determining the visibility of overlays and their linked views when filtering a certain difference type. For example when hiding a deleted node with a deleted edge or when hiding a moved child view inside a deleted node. The first case can also be implemented using the node and edge relations, but it has been decided to incorporate this into the overlay model as other models may require other dependency relations that are more complex to detect.

Some additional view components have been developed in Mervin that allow the visualization on top of the Papyrus models and were not supported by GMF, GEF or Papyrus. The most obvious one is the workbench component which is the component used by the controller assigned to the model review instance. It contains one or more diagram containers which can be shown in tabs or in resizable windows. The workbench manages the switching between those two different view modes as well as the window functions. Each diagram container mimics the root component used by Papyrus with several additions. Two layers are put above all other layers, one for the overlays and one for the off-screen indicators. Every overlay is placed using a layout algorithm that is defined by the controller of the overlays, as node and edge overlays must be handled differently, although they share the same parent component.

Another adaption implemented in Mervin was a more sophisticated handling of edge containment references. GMF places all edges within the same global layer, which also implies that they are drawn above all other elements. This causes problems with scrolled view components as edges are not correctly clipped. GMF and GEF handle these issues by hiding edges if one of its endpoints are not visible. But this also causes hiding of edges although they are partially visible. The problem often occurred in window mode and for long edges, so the decision was made to improve this issue. Mervin does that by assigning the edges to the first edge layer in the parent view component hierarchy that is also in the parent hierarchy of the attached view nodes. Some scrolled view components, like the diagram views in Mervin, already provide such a layer, and the normal clipping behavior of these view components can be used to draw edges correctly. However, this

only works for edges that are not assigned to view components within different scrolled view components, as they are drawn within the clipping region of another parent scrolled view component. On the other side, such cases did not occur with Papyrus models in the evaluation and during development.

Off-screen indicators use a radial projection algorithm to place the indicators at the borders of the diagram container. An *indicator merger* is notified permanently of layout changes and manages the display of the indicators. It creates a single merged indicator and hides the merged indicator if indicators overlap. But it is also responsible for disassembling merged indicators and make single indicators visible again if they do not overlap any more.

The workbench is also contained in a *focus viewport* which is used by the root controller of the unified difference view. This viewport provides the support for highlighting arbitrary components which are linked to views via controllers. It does that by drawing the diagram multiple times. First, it draws the whole diagram and adds a semitransparent viewport filling gray rectangle in the unified difference view. Afterwards, the diagram is drawn again for each focused component, but rendering is restricted to the a slightly larger area than the bounds of the focused figure. This way all other parts than the focused components appear grayed out, whereas the focused figures are rendered like before. Note that the diagram must be drawn with an opaque background in each draw call so that this approach works correctly. Which component is highlighted is the decision of the highlight listener of the unified difference view, which is further explained in Section 4.9.

Filtering overlays is implemented with respect to the visibility of overlays which the overlay depends on and which also affect the visibility of the component. If an overlay is not visible, also the dependent overlay is filtered from the view, regardless of the visibility state of the dependent overlay defined by the filters. Also overlays are only filtered from the view if all represented overlay information types are filtered, with the exception of deletion overlays with comments as described in Section 3.2.1.

4.8 Difference History

The implementation of the history of differences in the version history view was another challenge when implementing Mervin. EMF Compare provides a way to compute differences between two versions with or without a base version, but does not provide a way to track differences across multiple versions, and how they evolve. Determining the same difference or a similar difference in two comparisons is essential to visualize the evolution of a difference. This is exactly what the proposed visualization in Section 3.5 visualizes. Differences between the base version and the other versions, that occur in one or more versions should be shown in relation to similar differences when comparing the base version with another version. In fact there is a set of differences for each version, computed when comparing the base version and this version. Each of these differences in one set is linked together with all differences in the other sets via a similarity relationship. This relationship is expressed with a value between 0 and 1, where 0 means *not similar*,

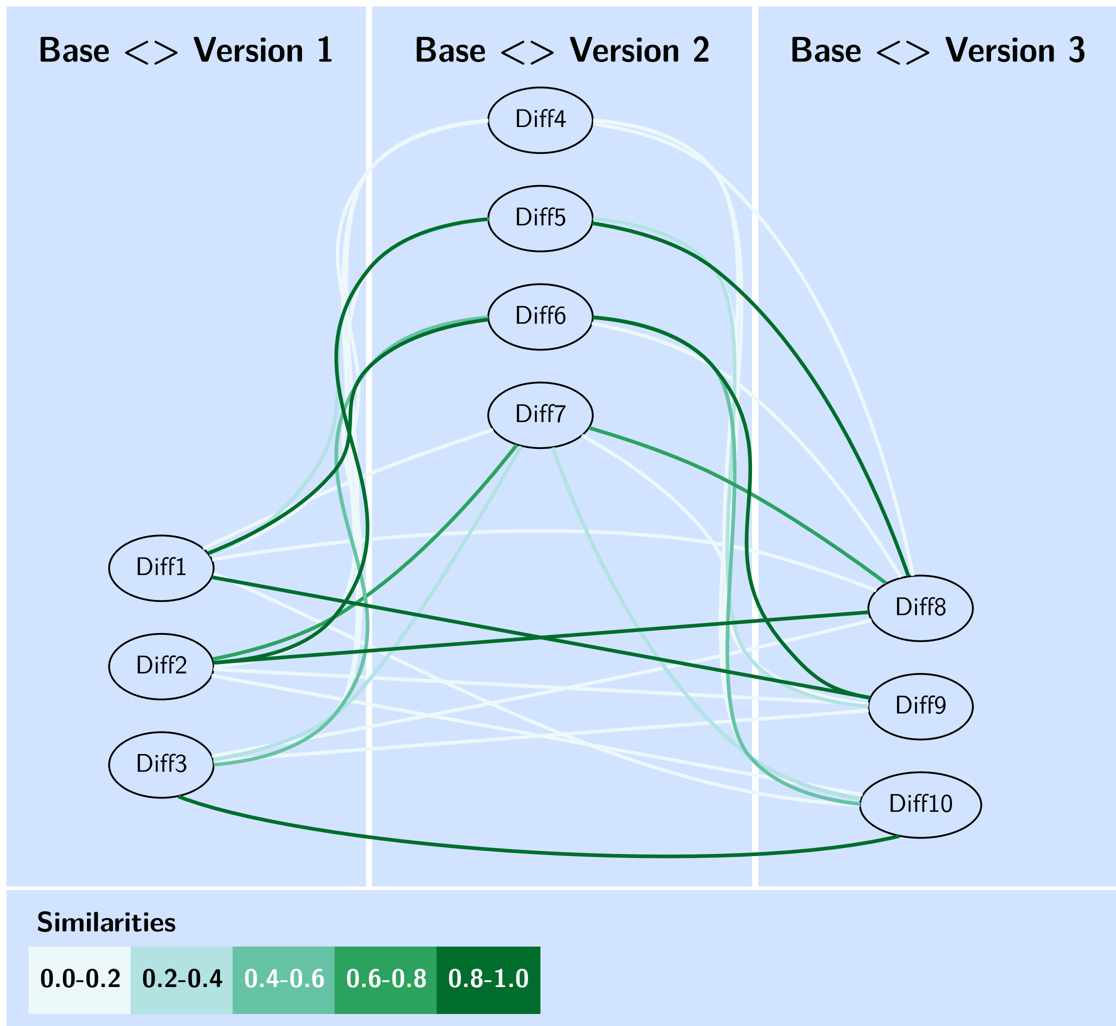


Figure 4.6: Example of similarity relations between differences. Similarity relationships are shown with colored edges, where the color indicates the similarity between the two differences.

and 1 means *equal*. As each difference in one set has a similarity relationship to every other difference in the other sets, a high number of relationships exist.

Evolution of differences is interpreted as a one to one relationship in this view, as the same difference cannot occur multiple times in the comparison of the base version with another version. There might be similar differences or no similar difference, but there is at most only one equal difference. So the most similar difference in the other set is considered as the most likely evolution of a difference. If more than one difference have the same similarity to another difference, one of them is chosen arbitrarily. As a consequence, the number of relationships vastly decreases as only one relationship exists

for each difference set and difference in a set. This is exactly what Mervin computes and uses to display the differences in this view.

Another important step is to compute the similarity. The first idea that has been implemented in Mervin was to re-use the EMF Compare matching mechanism and then use the distance used by EMF Compare. This is possible as differences in EMF Compare are elements of the comparison model which can be compared as every other model with EMF Compare. However, this yielded false matches as the internal model element ids used by the comparison framework where different between differences and EMF Compare had to use the edition distance instead. Apart from the changed elements, differences have only a few properties which might differ, which might be the reason for this behavior. It also seemed that the changed element had less influence on the distance as expected, although it is obviously very important when computing the similarity of differences.

So a specialized similarity metric has been used for Mervin. All differences of interest for Mervin store a value, the feature that has changed, as well as a difference type and the difference kind. The value is the element or value that has been changed on a particular feature of the changed element. Hence, the similarity metric is based on the weighted sum of the similarity of the difference value s_v, the similarity of the difference class type s_t and the similarity of the difference kind s_k. Equation 4.1 shows the metric as it is used by Mervin.

$$s = s_t * 0.25 + s_k * 0.25 + s_v * 0.5 \tag{4.1}$$

The weights are chosen arbitrarily, with the assumption that if the values of the differences differ a lot, the difference itself is also more likely to be not equal, and this should reflect in a smaller similarity value. Each similarity is a value in the interval $[0, 1]$ whereas the type similarity and the kind similarity are either 1 or 0, depending if they are equal or not. The value similarity is a little bit more complex and shown in equation 4.2.

$$s_v = max(s'_v, s''_v) * 0.9 + s_f * 0.1 \tag{4.2}$$

It is the weighted sum of the similarity of the maximum of two similarities s'_v and s''_v of the value, as well as the feature similarity s_f. Again, the weights have been chosen arbitrarily, with the assumption that the value similarity is more important. As mentioned before, multiple similarity values are considered in this equation. Mervin uses the *edition distance* provided by EMF Compare, which requires a comparison to look up other matches. In this case, the value might be contained in at least three different versions, one or two base versions, the version of the first difference and the version of the second difference. As a result four comparisons are possible to be used by in that case, the versions with their corresponding base versions, the base versions, or the versions itself. Mervin uses the comparison of the base versions and the comparisons of the versions itself, as the evolution between the two versions is of interest. Further research may check if the

other comparisons improve the results in this case. The edition distance is mapped onto the interval $[0, 1]$ before proceeding with the computation. The maximum has been chosen as the value might be only contained in one of the base versions or in one of the versions, for example if the value has been deleted or added. In the case of numeric values, the absolute difference between the two values is mapped to the interval $[1, 0]$. Other non-model values are simply checked for equality and 1 or 0 is used whether this check succeeded or not. The containing match has been explicitly excluded from the similarity, as differences that have been *moved* to other elements should be detectable as well.

Feature similarity is computed for mono-valued features simply by checking if the features are equal or not, which results in the similarity of 1 or 0. Multi-valued features are handled differently. Not only the feature itself can be different, also the position in the feature may be different. So the similarity is the absolute difference between the two positions, mapped from the interval $[0, MAXINTEGER]$ to $[1, 0]$. So the lower the difference between those indices, the higher the feature similarity for multi-valued features.

Now the question arises which differences to list in this view. Obviously, only the differences of one set can be listed, or all differences in all sets can be listed. However, this might result in a huge set of differences and some of the differences might be shown multiple times, as mentioned in Section 3.5. For example, a difference that occurs in n sets with a similarity of 1 is shown n times in this list. Moreover, these duplicates do not show up differently in the visualization, so it makes no sense to present them more than once to the user. So Mervin hides the duplicates by default, but provides a way to show them on demand. It should be noticed that this optimization might interfere with the grouping mechanism if at least one of the difference are assigned to a different group. Mervin provides currently no detection for that case.

Mervin groups differences by three different aspects. First they are diverted in differences detected in semantic models and in differences detected in notation models. EMF assigns model elements to packages and these packages are the next grouping criteria. The last criteria is the match that contains the difference. Therefore, equal differences might be contained in different groups, if the difference has been moved from one element to another between sets.

4.9 Highlight Contexts

Highlighting is another important aspect of the presented solution and the user interaction is described in Section 3.7. It was also mentioned that there are model specific options which elements are highlighted. But the process of highlighting in Mervin must be discussed before these options can be discussed for Papyrus models.

The process of highlighting is initiated by the user by specifying a set of elements to highlight, which are based on a selection in a view. These elements are passed to the

<table>
<tr><th>Type</th><th>Unified Differences</th><th>Review Explorer</th><th>Version History</th></tr>
<tr><td rowspan="2">Notation Element</td><td>Graphical Element</td><td>Parent Entries</td><td>Parent Entries</td></tr>
<tr><td>Tab/Window/Tray Title</td><td>Match Entry</td><td>Match Entry</td></tr>
<tr><td rowspan="2">Semantic Element</td><td>Graphical Elements</td><td>Parent Entries</td><td>Parent Entries</td></tr>
<tr><td>Tab/Window/Tray Title</td><td>Match Entry</td><td>Match Entry</td></tr>
<tr><td rowspan="2">Diff</td><td rowspan="2">Diff Value (recursive)</td><td>Parent Entries</td><td>Parent Entries</td></tr>
<tr><td>Diff Entry</td><td>Diff Entry</td></tr>
</table>

Table 4.1: Element types highlighted in each view based on the selected set of highlighted elements. Recursive means that the stated elements are evaluated for highlighting. Objects of types that are not mentioned are ignored.

highlight service which notifies any subscribers that listens for changes to the highlighted elements. It also stores the currently highlighted elements for any new subscribers. Each subscriber may react to the new set of highlighted elements and highlight parts that are associated with the highlighted element. A subscriber is a view in Mervin, but other elements may also subscribe to the *highlight service*.

It is important to note that the selection of the elements is done in a different context as the actual highlight. So, an element type might be selected for highlighting that is never displayed in the highlighting view. For example, a semantic model element is per definition not present in the version history view or in the unified difference view. But their linked differences or notation model element are present in the view. On the other side, the user might want to explicitly highlight only the corresponding notation model element for a selected semantic model element. The latter must be done in the context of the view that triggers the highlight, the former is the decision of the view that highlights the element.

Initial testing during development showed that being more restrictive while highlighting associated elements in a view is a better strategy than highlighting all associated elements. It enables fine grained highlights, while the user may trigger highlights for the associated elements from other views. While this is the case, highlighting no associated elements might result in no highlights at all, which should be avoided. Additionally, some elements cannot be selected in other views or are special elements only known to the view, which should be highlighted. A tab that contains an highlighted element in the unified difference view is an example for that case. An overview of which types of selected highlighted elements trigger which highlights in which view are shown in table 4.1. It has to be noted that notation model elements are always resolved with consideration of the unified model map in the unified difference view, as copied notation model elements might be selected for highlighting.

Selecting elements to highlight in Mervin views is done either by selecting a single element in the view, or by triggering a set of derived elements from the current selection. The former might include the reduction of the selected element to a set of commonly known types across the views. Table 4.2 shows an overview of the reduced types. Context menus

View	Selected Type	Passed Element
Unified Difference View	Graphical Element	Notation Element (copy) Notation Element Semantic Element
Version History View	Match	New Element Old Element
	Diff With Similarity	Diff
	Entry	Version Values
	Named Entry	All Subentry Selections (recursive)
	Object Entry	All Matches Selections (recursive)
	Other Element	Same Element
Review Explorer View	Match	New Element New Element
	Other Element	Same Element

Table 4.2: Elements passed to the highlight service from each view. Recursive means that the stated elements are reduced for selection.

trigger the other case, which are not shown if they cannot be done for the currently selected element. Mervin supports the following selection actions: Selecting the values of differences, the notation model views that reference a semantic model element, the semantic model element referenced by a notation model view and the differences that reference a model element. Again, copied notation model elements need to be considered in the unified difference view. The original elements as well as the copied notation model elements are passed to the *selection service*. Selections from the comment view are straightforward, as comments can only be linked to notation model or semantic model elements.

CHAPTER 5

Evaluation

Visualization of graphical models and diagrams in the context of model review is a complex topic and the preceding chapters described the main problems in detail. The previously presented prototype implemented the proposed solution and a case study been made to evaluate the proposed solution. The case study has been created based on the methodology presented by Lee [Lee89]. To the best of our knowing, no specific tool exists that supports a similar review process for diagrams and graphical models as the code review tools presented in Section 2.1. However, it is possible to use a combination of tools, a *tool set*, to do a similar review with the same models and review data that the prototype supports. Such a tool set is currently also the only option to do a review with diagrams and graphical models. The tool set contains of components that interact with the review data, view the diagrams and graphical models, and compare them. The prototype *Mervin* is based on Gerrit for the review data, so Gerrit, EGit and Mylyn have been used extract the code review data from the review repository in the tool set. *Mervin* supports Papyrus UML models which can be shown by the Papyrus editor and compared using EMF Compare. Therefore Papyrus and EMF Compare is used in the other tool set. Most of these tools are integrated within an application based on the Eclipse Platform, such that the participants of the case study did not have to switch between multiple applications. As a result, two rivaling theories have been defined:

Theory 1: The combination of EMF Compare, Papyrus, Mylyn and Gerrit support users better in the review process than Mervin.

Theory 2: Mervin supports users better in the review process than the combination of EMF Compare, Papyrus, Mylyn and Gerrit.

Of course, both theories can only be evaluated for a given audience in a case study, which is described in Section 5.2. Tool support is defined in this evaluation by the following factors, all in the context of graphical model review: Identification of issues, verification

of reported issues, identification of differences, tracking of differences across multiple versions, identification of linked elements within a change, navigation through linked elements within a change, and tracking of issue discussions across multiple versions.

Identification of issues in a particular version is obviously a key part of the reviewing process and is dependent on the experience of the reviewer. The verification of reported issues is related to the identification of issues, but more complex. In this case reviewers need to understand the reported issue and validate it. If the issue is justified in the opinion of the reviewer, it can be validated in the context of the another version to check if it also occurs in that version.

In order to do that, reviewers have to be able to identify differences between versions as issues obviously arise only if elements are changed. Some sort of tracking of the differences is also needed as issues might get resolved by adding or eliminating differences. Issues and elements are not necessarily part of every version in a change, so reviewers have to keep in mind linked or related elements within a change. Supporting the navigation through those elements might be of use while understanding the artifacts and the elements they contain. Issues are not always justified and need to be discussed. Reviewer need to be aware of these discussions even if the issues are resolved in other versions, at least to verify that all valid aspects of the issues have been resolved.

A number of predictions have been defined for this evaluation based on the previously defined factors. They are based on the assumption that one tool set outperforms the other with respect to one of the factors. If the majority of these predictions is true for a given tool set, the corresponding theory will be seen as confirmed for the evaluated audience. Every prediction is evaluated with a prepared change and a typical review task that the participant has to do during the case study. The completion time for subtasks is not measured, as the participants should solve the tasks with the tool sets on their own, without exact instructions on how to proceed and in their own pace. A feedback session is also part of the setup and will be also considered when evaluating all predictions. A detailed explanation of the evaluation setup is given in Section 5.1.

Prediction 1: *If users miss less issues with one tool set, this tool set is considered as supporting the user in finding issues better than the other.* For a given change with issues, the number of missed issues gives an measurable number which can be easily obtained. Moreover, issues are categorized which may provide more detailed information which types of issues have been found and which have not been found. Due to the fact that a prepared change might contain issues that have not been foreseen, also unexpected issues arise during the evaluation which might be justified or not. These issues are noted and count to the total number of issues at the end of the case study. So the missed issues contain of the number of missed expected issues and the number of missed unexpected issues.

Prediction 2: *If users are able to verify reported issues easier with one tool set than the other, the former tool set is considered as supporting the verifiability of issues better*

than the other tool set. This prediction is verified directly by preparing a change with identified justified or unjustified issues in one version and the user acts as contributor. Participants of the study have to interpret and evaluate the comments if the reported issues are justified or not. The number of correct answers can be measured and compared. However, the decision if an issue is justified is not always clear and participants might understand the issue, regardless of the expected answer. So participants are also asked to explain the issue and their decision to give further insight if the answer does not match the expected answer. The issue is considered to be verifiable with the tool set if the explanation shows that the participant did not misunderstood the issue. Another indirect assessment of this prediction is observation. Either while participants evaluate whether an issue is justified or not, or while participants review a prepared change with multiple versions and reported issues. The missed issue count in the latter case also gives a hint that issues could not be verified.

Prediction 3: *If users are able to identify differences easier with one tool set than the other the former tool set is considered as supporting the difference identification better than the other tool set.* Participants are observed during the review of a change to verify this prediction, and by evaluation of the explanations of the participants on issue detection and verification tasks.

Prediction 4: *If users are able to track differences easier with one tool set than the other, the former tool set is considered as supporting the tracking of differences across all patch sets better than the other tool set.* Again, the behavior of the participants is observed during all tasks to verify this prediction. As mentioned earlier, issues are categorized, and one issue category is the *recurring issue*. These are issues that have been reported in one version, resolved in the subsequent version and recur with the same differences in the version after. The number of missed recurring issues can be used to measure and compare tool sets with respect to this prediction.

Prediction 5: *If users are able to identify the linked elements of change elements easier with one tool set than the other, the former tool set is considered as supporting Identification of linked elements better than the other tool set.* As most of the predictions, observation of the participants is also needed here to validate this prediction. Additionally, the answers and explanations of the participants are also used to evaluate and show shortcomings in the tool set with respect to identify linked elements.

Prediction 6: *If users are able to navigate through linked change elements easier with one tool set than the other, the former tool set is considered as supporting navigation of linked elements better than the other tool set.* Navigation is a done by the participant and can only be observed while the participant solves the given tasks. So observation during the tasks is done to verify this prediction.

Prediction 7: *If users are able to track discussions about issues across versions easier with one tool set than the other, the former tool set is considered as supporting tracking of discussions about issues better than the other tool set.* Without restricting the participant to a particular workflow, arguments abouts identified issues are evaluated and participants are observed during the tasks to verify this prediction. Moreover, the number of missed reported issues is also used to verify this prediction. Additionally, a single question about a discussion detail is asked after the review task to verify if the participant was aware of the discussion.

5.1 Evaluation Setup

As a result of the theories mentioned in the previous section, the case study was done with two different tool sets: The developed prototype of the proposed solution called *Mervin* and the combination of EMF Compare, Papyrus, Mylyn, Gerrit, and EGit, called the *old tool set*. These tool sets were used by participants to solve a main task in typical review scenarios. A scenario description with the requirements in written form was given to the participants before each task. Each participant was given a short time before the tool sets have been used to read and ask questions about the task. Participants are also allowed to ask questions about the tools and UML notation during the task, but questions regarding the semantics of elements of the change or about issues and comments are not answered. Besides answering those questions, the interviewer also observes how the participants interact with the tool set. The prepared changes are made out of a single Papyrus UML model with four different diagrams. Which of these diagrams had differences depends on the task, but always more than one diagram is affected by the scenario description. So it is in the participants duty to decide which diagram needs to be reviewed.

In the first scenario, the participants were asked to review the last, not reviewed version of a change. They have to decide if they would accept the version, identify issues, and argument why these issues are valid. A change in this scenario contains multiple versions with prepared issues, that evolve across the versions. It contains four expected issues, each in one of these issue categories: recurring issues, unreported missing requirements, obvious unacceptable errors, and reported unresolved issues. Furthermore, a fifth category contains all unexpected issues reported by all participants. Recurring issues are issues that have been reported, resolved and recur in the version to review. Unreported missing requirements are requirements that have been explicitly mentioned in the task description, but have not been implemented in the model. Obvious unacceptable errors are issues that are easy to spot when reading the model in its graphical form, like a missing control flow between two actions in an activity diagram. Although this issue category can be seen as a missing requirement issue, an own category has been created for these type of errors as it is expected that such errors are harder to see in non-graphical representations of graphical models. The last expected issue category contains issues that have been reported in previous versions and have not been resolved in the version to review. After the participant finished this scenario, a quick question about a comment to an issue in a

previous version was made in order to take a spot example if they missed the comment or not.

Afterwards, participants switched to the role of an contributor who has to interpret the result of a reviewed version in a different change. Such a change contained only one version with comments about two issues: One is expected to the justified, the other is expected to be unjustified with a given reason. Participants also had to identify which issues have been reported by the reviewer and had to decide which of them are justified or not. More importantly, they also had to explain why they think that an issue is justified or not.

These two scenarios have been evaluated two times for each participant, each time with a different tool set and different prepared change. The prepared changes could be used with each tool set and have been chosen in a way that each change was used the same number of times with each tool set during the whole case study. However, no change has been reused with the same participant for a given scenario, as they were already aware of issues in these changes. Possible varying difficulty levels of prepared changes cannot be ruled out due to the size of the models and the traceability of the scenario description. So this switching of the input data was necessary to avoid side effects on the results. During the review, two changes for each scenario have been prepared for two case examples, and each change in the case example used the same base version: Models used by a warehouse to describe their workflows, and models used by an event management agency for managing their workflows in their software.

Switching tool sets and changes requires that the participants get some time to accommodate to the changed situation. Therefore, tool sets and case examples were only switched once, and the users have been introduced to the tool sets right before starting the scenarios. Moreover, they also got a couple of minutes to make themselves familiar with the base model version of the case example. In order to verify that they were able to use the tools and extract basic information about the change, participants have also been asked some spot questions about some properties of the change before the scenarios started.

Before the scenarios have been started, each participant was introduced to the review process and the concepts of model review if necessary. Also some questions regarding the background knowledge about code review and modeling tools were asked. A feedback session was done after all scenarios have been completed. In this session subjective impressions of the participants have been noted and they have been able to give their own opinion about the tool sets and the features they liked or disliked.

The main audience of participants for this case study are persons who are familiar to UML, are familiar to how code or model reviews work, and optionally are familiar to Model-driven engineering. To avoid effects of prior knowledge, participants should have not much experience with versioning of Papyrus models.

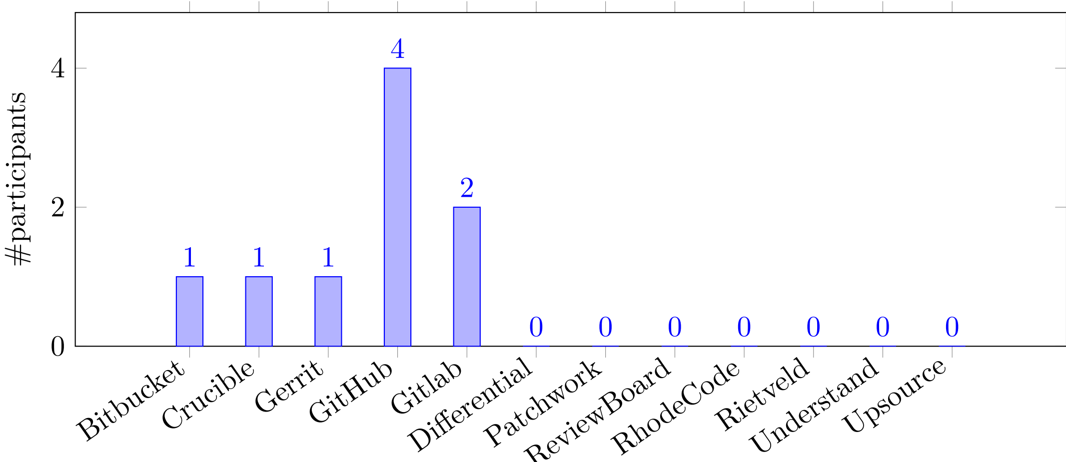
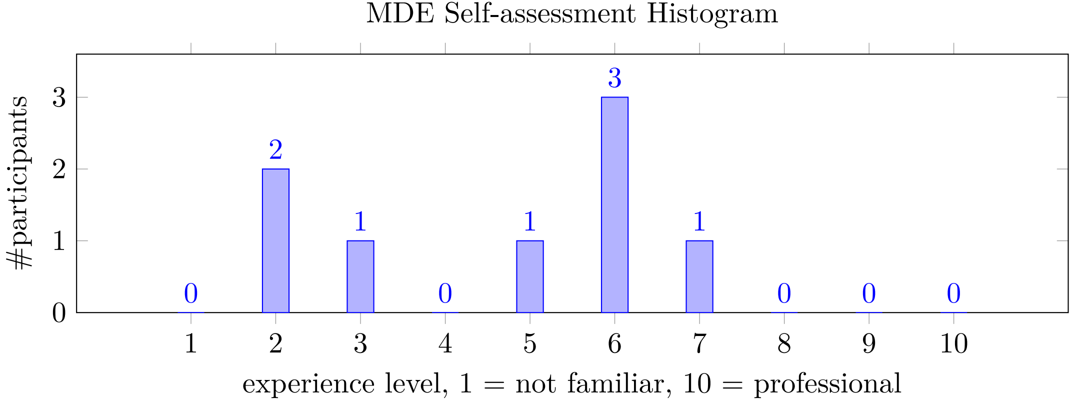


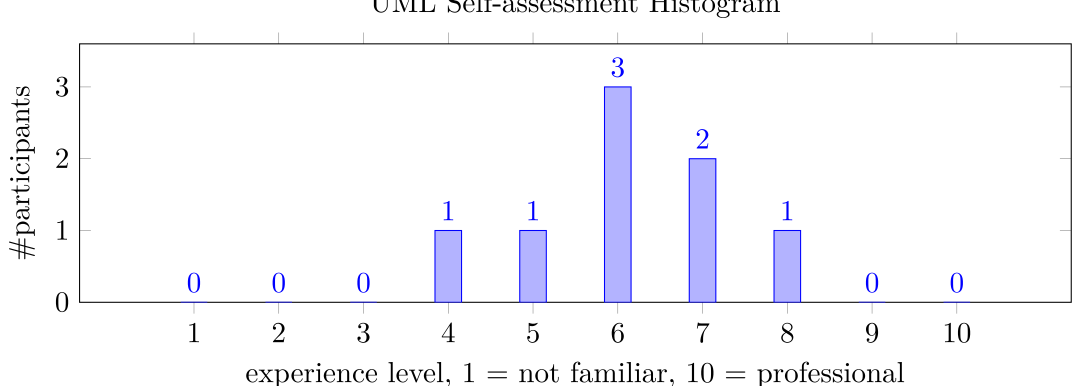
Figure 5.1: Histogram of code review tool used by the participants prior to this evaluation.

5.2 Evaluation Results

Eight participants took part of the case study and therefore the scenarios in the two case examples have been executed four times with each tool set. Most of the participants already used a code review tool, whereas two participants never used a code review tool before. The code review component of GitHub has been used by half of all participants and was therefore the mostly used code review tool among the them. Prior knowledge was evaluated on a self-assessment scale from one to ten, where one means no experience and ten means a lot of experience in the given field. All participants had at least some knowledge Model-driven engineering, in contrast to experience with Papyrus, where just one user had some experience. An additional question revealed that this participant had no knowledge about the internal structure and versioning of Papyrus models, so it is not expected that this will interfere with the precondition defined in Section 5.1. However, all participants reported enough experience with UML such that they were able to interpret the prepared models without additional instructions.

In the first scenario, participants had to review a version of a prepared change, and the most obvious metric is the missed issue error rate. These are compared between the results of the two case examples to rule out side effects due to varying difficulty levels. As a result, it can be observed that the error rate of missed issues decreased by $11,11\%$ in the event management example, whereas it decreased by $25,0\%$ in the warehouse case example. It has to be noted that more than the half of the existing issues have been missed in the event management case example regardless of the tool set. In the case of the warehouse example, also more than the half of the existing issues have not been found with the old tool set. In contrast, the proposed solution was able to reduce this error for the warehouse example. So in general it can be said that the error rate of missed issues is reduced when participants used Mervin, although the improvement keeps within limits.





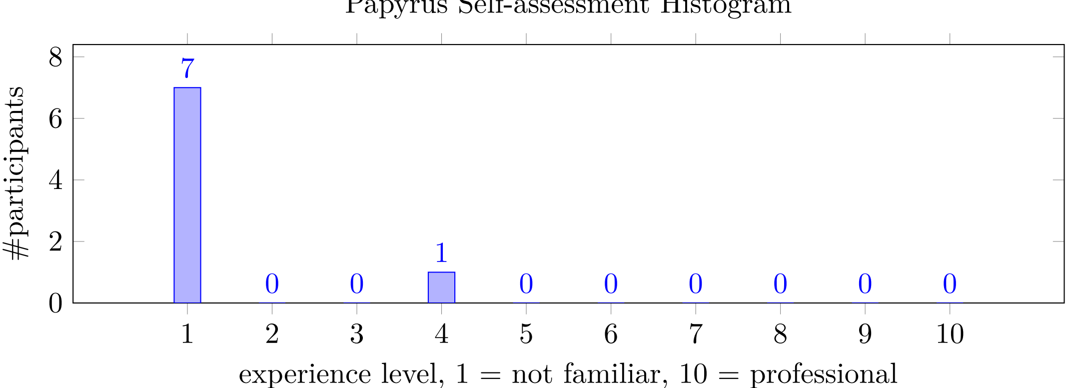


Figure 5.2: Histogram of the self-assessment of the participants prior knowledge with Model-driven engineering, UML and Papyrus, where one stand for *not familiar* and ten stands for *professional*.

A closer look at the error rate by issue category reveals that the missed issue error rates decrease in the recurring issue category in both case examples when using Mervin. Again, also in this case the error rate is far less in the warehouse case example than in the other case example: No recurring issue has been missed by participants in the warehouse case example using Mervin, while half the participants using Mervin with the event management example missed the recurring issues. No difference can be seen in the error rates for the unreported missing requirement and the obvious unacceptable issue categories. The last expected issue category yields contradictory results when comparing the two case examples. While it increased by 50% when using Mervin in the event management example, it decreased when using Mervin in the warehouse example.

Another interesting point is that the participants reported five unexpected issues for the first case example and only one for the warehouse case example. Although this was the case, participants where not able to find those unexpected issues with the old tool set in the event management case example. On the other hand, no difference could observed in the error rate for the unexpected issues in the warehouse example. But this might be due to the low number of unexpected issues and might be of interest for further research.

No difference can be reported in the number of incorrectly reported issues: Two participants reported incorrect issues in the event management case example, but both used different tool sets. In the warehouse example, no such issues have been found with both tools. The spot example on the issue comment revealed that two participants missed the comment in the warehouse case with the old tool set, whereas this was not observed in the other case example. Although this is only observed in one example, it should be noted that Mervin had a positive impact in this case.

The measurable results from the second scenario show no impacts of the used tool set on the contributor tasks. All issues have been correctly identified and understood, regardless of the tool set. The only difference were that the participants gave different reasons why the issues are justified or not. Nonetheless, all given reasons were valid.

More differences between both tool sets have been observed on the user behavior. Three participants had problems reading the differences from the comparison view of the old tool set in certain cases. Two reported them directly to the interviewer during the both scenarios. Another one searched for an reported issue in the differences of a completely different and unrelated diagram in the contributor scenario. However, this participant noticed his mistake after some time searching for the issue, and was able to give the correct answer after some time. Only one participant used only the comparison view of EMF Compare to view the differences. All other participants also used the Papyrus editor to view a particular version of the UML diagrams, often in combination with the Mylyn view that showed the comments. But sometimes the Papyrus editor was also used in combination with the comparison view, although it supported the display of graphical elements of the diagrams, when selecting the proper elements in the notation model. This might also be impacted by the fact that the graphical display of the compare view did not render the diagrams properly in some cases. Most of these display bugs could be solved with the two clicks or reloading the compare view, and the interviewers made

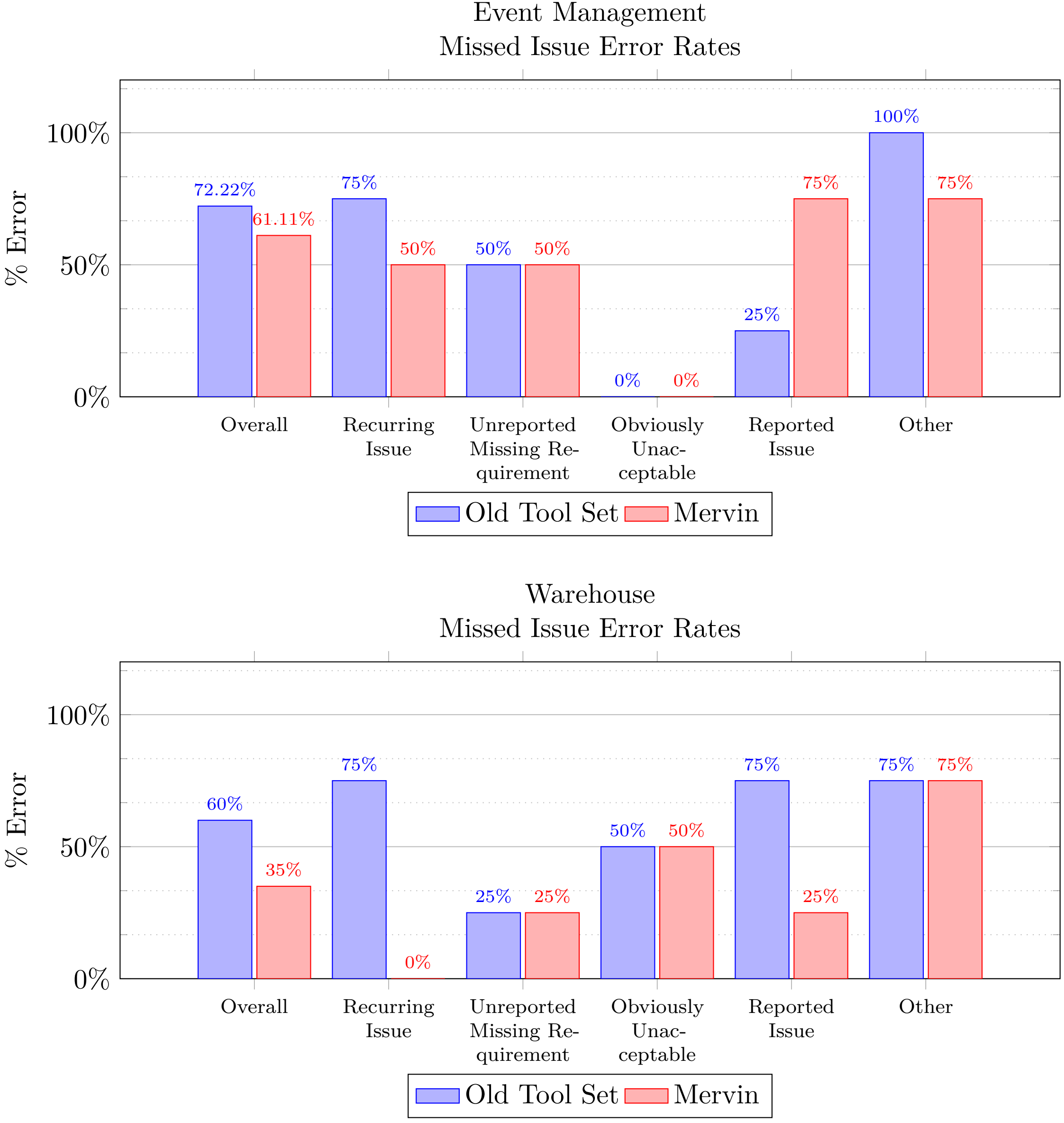


Figure 5.3: Missed issue error rates.

the participants aware of the workaround if a display bug occurred. One participant also ignored the comparison view completely and used only the Papyrus editor for the scenarios. The Papyrus editor does not display differences, and so this participant also explained to the interviewer that one diagram has not been changed although it has been changed.

The general workflow during the review scenario with the old tool set was mostly similar across all participants, except for the one participant who used only the Papyrus editor: First the task description has been read, then the comparisons are read for some versions with their comments. The Papyrus editor has been used sometimes between the comparisons, as well as the task description to check the requirements. Not all versions have been viewed by all participants, one skipped one version, two participants ignored all versions except the version to to review and the base version. It should be noted that the comments to all versions have been read at once by all participants. Another interesting note is that no participant switched back to a comparison that has been viewed before. Instead, a comparison has been often read completely by stepping through each difference before switching to the next. A few of the participants only stepped through the differences of the notation model. Layout differences have been quickly identified and stepped over by two participants.

In contrast, participants switched more often between various comparisons of versions while they used Mervin. They mostly did not step through the differences, but switched between the diagrams to read the differences of particular versions. No participant reported confusion, although they used and switched between multiple views while reading the differences between versions. As expected, the most used views were the unified difference view and the comment view. Participants usually read the differences of a version before they read the comments in the comment view. The temporary and permanent highlight-feature of the comment links have been used by all participants and they sometimes also switched back to the unified view and read the related elements. Layout difference overlays have been hidden by half of the participants and the option to display the deleted elements has not been used very frequently. The window mode switch has not been used by any participant. Another view that has been used frequently was the review explorer, although it was remarkable less frequently used than the unified view and comment view. It has been mainly used by the participants to identify changed diagrams. Those participants who wanted to step through the differences like they did in the old tool set, used the review explorer instead. Although Mervin provided also a list of all differences in the version history view, this view was not used for this purpose. Additionally, the version history view was only rarely used, and most participants were not able to make use of it for both scenarios. Similarly, the property difference view was also barely used. This might be due to the fact that it contained very fine grained details of the models that are not of use for users without expert experience with Papyrus models. In fact, some participants even minimized those two views to get more screen space for the other views.

The second scenario contained only the base version and the changed version with a

Figure 5.4: Subjective impressions of the participants regarding which tool set provides better support with respect to a particular subject.

different task, and the participants also used both tool sets slightly differently to solve it. Five users relied only on the Papyrus editor in combination with the Mylyn editor for the second scenario and the old tool set, ignoring the actual differences between the base version and the other version. One user also used the Mylyn view with the comments on a second screen, to avoid switching between views. The interviewer also had the subjective impression that it always took some time before the participants identified the elements in the diagrams that have been mentioned in the reported issues.

In contrast, the participants always used the comparison of the base and the first version while using Mervin in this scenario. All participants also used the highlight-feature of comment links and were able to quickly navigate to the linked elements. More than the half of the participants used this feature on all links in the comments.

The subjective impressions by the users give a clear indication that most of the participants preferred Mervin over the old tool set. Each participant has been asked which tool set provides better support with respect to a particular subject, and the exact results are shown in figure 5.4. The subjects have been the *clarity* and the *comprehensibility* of issues and changes between versions. Clarity describes how easy it was for the participant to find and identify issues and differences. Whereas comprehensibility describes which tool set provides the better support to identify why differences and issues exist and how they are related to other elements.

Besides that, the participants also gave feedback about Mervin with respect to the old tool set. With the exception of one participant, all participants reported that the linking feature of comments to elements in combination with the highlight-feature was the most liked feature. Also six participants reported that the unified difference view was much

more intuitive and easier to read compared to the comparison view of the old tool set. Half of the participants noted that the presentation of the differences was clearer than in the old tool set. Two participants said that Mervin provided less information overload and therefore easier identification of differences. Moreover, two of them liked the difference overview of the model provided by the review explorer, which is not present in the old tool set. A few also mentioned the always present presentation of comments in the side by side arrangement and the different alignment of comments based on the author. The difference overlays, overlay filtering, off-screen indicators and the predefined comparisons in the review explorer have also been mentioned.

Some features have not been used and this also reflects in the feedback of the disliked features of Mervin: The version history view and the property difference view have been reported as unnecessary by two participants. Moreover, one participant noted that the version history was too complicated and hard to understand. Also the interaction of the version selection was not optimal according to two participants: The new and old version where two separate input fields, and changing one of them caused a recalculation of the unified difference view. So switching both sides to completely different version causes two total recalculations of the unified difference views which disturbed the workflow. Two user interface bugs have also been reported by one person: Every recalculation caused a different order of the diagram tabs in the unified difference view, and empty comment groups have not been removed from the comment view when switching versions. Another one noted that is was not clear that the comment column order reflect the selected comparison scope and that the highlight of tabs was hard to see. One also disliked the overlays and suggested to color the elements instead.

More than the half of the participants, five in fact, requested that also comments out of the comparison scope should be shown in the comment view. Three of them also requested that a chronological view of the comments would also improve the comment view. Comments are currently only sorted by their creation time within a group. Another requested feature was to quickly navigate from a selected element to a linked comment, which was not supported in the prototype, but can be implemented with the highlight-feature. Stepping through comments has also been requested by two persons. Another one suggested to support marking of comments as resolved as well as collapsing comment replies or comment groups to save space. Also more user interaction suggestions have been reported by various participants: For example, the review explorer should also be able to open a diagram in the unified difference view with a double click. Moreover, the highlight-feature should be adapted to quickly focus the corresponding element in another view, especially in the unified difference view. The possibility to zoom the diagram was also requested, as well as the support to quickly shift the comparison scope back or forth. This shift was proposed only for comparison scopes were each version is the predecessor or successor of the other, as this operation must be defined for other configurations. It should also be noted that some participants said that, according to their opinion, versions within a change are understood as iterative improvement, instead of different versions based on one base version. This coincides with the request to shift the comparison scope,

so it might be of interest for further research to provide a different comparison scope selection method which combines both operations.

Based on the previously presented results and observations, the predictions and the theories defined at the beginning of this chapter can be evaluated. *Prediction one* is evaluated using the missed issue count and is therefore true for Mervin, but not for the old tool set. The second scenario showed no measurable difference between both tool sets. But observations showed that a few of the participants had initial problems to relate elements, comments and issues. That also reflects in the subjective impressions, where almost all participants preferred the presentation of Mervin with respect to the comprehensibility of differences and issues. As a result, the *second prediction* is also true for Mervin and false for the old tool set.

The subjective impressions of the participants on the clarity of differences also gives a strong argument for Mervin for *prediction three*. Also observations showed that three participants had problems while reading the differences. Moreover, the frequent use of the Papyrus editor also gives a hint that the comparison view of the old tool set does not provide all information for the participants to understand the context of the difference. Prediction three is therefore true for Mervin and not for the old tool set.

Participants missed less recurring issues with Mervin than with the old tool set. As a result, *prediction four* is clearly true for Mervin, but not for the old tool set. *Prediction five* is true for Mervin based on the subjective impressions reported by the users and on the observations made by the interviewer. Linked elements are usually part of the description of issues and the participants provided a clear feedback on this aspect: Mervin provides a better clarity of issues and a better comprehensibility of issues compared to the old tool set. Also, the interviewer observed that participants took some time when searching for elements mentioned in a comment when using the old tool set. This may be due to the fact that the participants had no explicit list of linked elements and had to decide on their own if an element is mentioned in the comment. On the other side, Mervin's comment links, the comment overlay and the temporary highlight-feature was extensively used by the participants to quickly identify if an element is linked to a comment and which are the linked elements of an comment. This also coincides with the fact that this feature was the most liked feature of Mervin. *Prediction six* is also true based on these observations and for similar reasons. Again, Mervin's comment links in combination with the highlight-feature was used by all participants to quickly navigate between the views. On the other hand, the old tool set did not provide an explicit way to quickly navigate to linked elements in the other views. So participants switched between different views, and tried to find the corresponding elements on their own.

No particular behavior of the participants could be observed for one of the tool sets that allows to verify *prediction seven*. The only observation was that two participants missed comments about issues with the old tool set in one case example. Apart from that, the number of missed reported issues are contradictory across both case examples. Therefore this prediction cannot be clearly evaluated to be true of one of the two tool sets. However, all other predictions are true for Mervin, so *theory two* has been verified

for the evaluated audience. As a result, a strong indication is given that the proposed visualization solution improves the visualization of graphical models and diagrams in model reviews.

5.3 Summary

In summary, a positive impact of the proposed visualizations has been observed in the presented evaluation. The theory that Mervin supports the review process better than the other tool set has been verified by seven predictions where the majority was true for Mervin, but not for the other tool set. So most of the proposed visualization techniques can be considered as an improvement to the current state of the art.

However, it also shows that there is room for improvement, and some parts need to reworked. Especially the high error rates found in the evaluation results give a strong indication that the current support for graphical model review in Mervin and the other tool set requires more improvement. Two of the views of the proposed solution where not frequently used, so they might need to be reworked. Especially the version history view needs to be improved or be replaced with a more intuitive visualization, as it requires the most complex data set, the difference matching information across all versions. The view coordination using highlights can be clearly seen as a success, as it was the feature with the most positive feedback from the case study participants. It was mostly used for linked elements in comments and for navigation between views. Participants also proposed some improvements on some of the views. One of the most requested improvements regarded the comment view, which should show all comments, rather than showing only the comments that are part of the current comparison scope.

CHAPTER 6

Conclusion

As presented in this thesis, visualization of evolving graphical models and diagrams in the context of model review is a complex and a non-trivial task. It has been shown that typical code review visualizations exists but are not easily adaptable for graphical model review. Moreover, existing model comparison tools exist, but lack of explicit model review support. A set of various visualization techniques have been proposed and combined to a single proposed solution that aims at improving the support for graphical model review. This solution has been implemented in a prototype application which is called Mervin, and is freely available under an open source license. It supports graphical model review of Papyrus UML diagrams with Gerrit. Implementation details of the prototype have been discussed to give an example on how to apply the proposed solution on existing frameworks and graphical models. Also implementation problems have been discussed, and in the case of GMF some problems could not be solved without additional information or in-depth knowledge of the graphical model's implementation. Afterwards, this prototype has been evaluated with a case study of eight participants, where their performance and subjective impressions have been used to confirm one of two rivaling theories. The positive impact of the proposed visualizations has been confirmed in this case study. In particular, the unified view and the view coordination with highlights received the most positive feedback from the case study participants. The following sections give prospects on future research and a summary of the findings.

6.1 Future Work

As mentioned in Section 3, understanding the reason of the change is a topic that is neither addressed by the proposed solution nor by other code review tools. This might be due to the fact that all tools store only the changes and only a textual description can be given by the contributors. However, future research may suggest other forms of

descriptions of the reason or may provide methods to deduce reasons from the description and the differences in the change.

Another aspect that might be of interest of future research is whether the usage of only explicit layout information has an impact on how the proposed solution is understood by users. The prototype implementation is currently based on Papyrus diagrams which relies mostly on implicit layout information for the unified graphical model as well as for visualizing the previous layout. Doing so raised problems and restricted the amount of layout information that could be extracted as described in Section 4.6. Developing another prototype based on explicit layout was out of the scope of this thesis and so it is unknown if this has a positive or negative impact on the results presented in Section 5.2. The same is also true for implementations that are more specialized for certain graphical models an therefore have more predictable information about the layout and the layout algorithms used to render the graphical model.

As mentioned in Section 3.6, other metrics can be shown in the review explorer. A wide variety of metrics are possible, and it is not clear which of them may be of use during graphical model review. Viewing all of them would obviously require too much space and therefore only a small subset of metrics must be defined. It may also be true that various different metrics are only of use for certain models. Specifying such a set of useful metrics for this view in combination with different models might be an interesting topic for future research.

The unified difference view also posed some open questions that might be of interest of future work and most of them are caused by the merging of both model versions. The most obvious one is the layout adjustment, especially whether a particular layout adjustment outperforms the no explicit layout adjustment method proposed in this thesis. This adjustment must allow judgment of the real layout, as this is also a valid reason to reject a model or a diagram if the layout is defined by the user. Another aspect that might be of interest for future work is how to handle the deletion inconsistency problem mentioned in Section 3.2.2.

As mentioned in Section 3.2.1, showing layout outlines require also the visualization of related elements. However, besides relations defined in the presented structure of graphical elements, also other relations of the model may be of interest in this case. This depends on the used model and should be the scope of future work for specific models.

The comment view is proposed with a simple grouping algorithm that is based on the connected components of graph spanned by comment links. This works well if no further information about the model is known, but for specific models more detailed grouping algorithms can be defined. Moreover, this simple grouping mechanism might also group comments which are not semantically related to each other. Future research may answer the question if this is the case and if it can be improved for general or specific models.

View coordination provides a method to highlight derived elements from the current selection. Which elements are of use for the user depends obviously on the used model, as it has been done in the prototype implementation for Papyrus models. So future

research may answer the question which elements are of use for other models. Another open question is how to order highlighted elements in the unified difference view, to allow stepping through all highlighted elements. As mentioned before, the reading direction or semantic based navigation would be an quick guess, but it remains to be proven if this is true.

The evaluation results showed that users using the proposed solution were able to find more unexpected issues in one case example than with the old tool set. This might be a side effect of the case example or an indication that the proposed solution supported the detection of additional issues. To prove that, further investigation with other case examples and bigger case studies with more participants are needed.

Also the participant feedback suggests that the comparison scope selection might be reworked. Most of them navigated through the versions from the first to the last version in a sequential manner. This can be implemented as a comparison scope shift back and forth. Future work may prove if this would be an improvement for users than the manual section of the versions that has been implemented in the prototype.

The last suggestion would be to research whether the notation model and semantic model separation of Papyrus models should be shown by default to average users or if it makes sense to hide it until requested by the user. During the evaluation, most participants seemed to be overwhelmed by the differences in both models, which where in fact duplicated, as each UML element also had a view element in the notation model. Moreover, the relationship between the UML element and view element was not visualized in the used version of the EMF Compare view, and only the selection of a notation difference showed the graphical presentation of the difference. However, this was only a subjective impression by the interviewer.

6.2 Summary

Multiple linked views with different visualization techniques have been proposed to solve the initially defined problems of visualizing differences during graphical model review. This proposed solution requires a complex set of input data. As multiple versions are present in a change, comparison data of all versions is needed. At least the comparison data needed by the review explorer must be precomputed and stored during the whole review, other comparisons can be computed on the fly when needed. Moreover, the version history requires the matching information of elements with differences across all versions during the whole review. Modeling frameworks must provide a way to draw on top of graphical models and extract the layout information of elements, as well as the possibility to reference elements within them. Review frameworks must provide a way to store comments with links to model elements. The developed prototype Mervin showed that it is possible to obtain this complex input data set and use existing modeling and review frameworks to implement the proposed visualization solution.

Using a unified difference view instead of a side by side view resulted in positive feedback

by the participants in the case study. No explicit layout adjustment is proposed to avoid misunderstandings by the user and to also allow reviewing of the layout of a graphical model. But this comes at the cost of the problem on how to deal with overlapping elements caused by deleted elements. Filtering and interaction was used in this solution, but future research may provide better solution for this problem. Interactive overlays have been used in favor of simple coloring of elements to avoid conflicts with semantic coloring in the model syntax. Layout differences have been segregated from atomic differences, as they might be of interest in some cases and for some not. Moreover, these differences can be visualized on demand directly in the unified difference view.

The positive impact of the proposed solution have been discussed in Section 5.2 and Section 5.3, so most of the proposed visualization techniques can be considered as an improvement to the current state of the art. Although this is the case, the high error rates found in the evaluation results give a strong indication that the current support for graphical model review might need more improvement. The results also show that some views of the proposed solution might need to be reworked. Feedback of the participants shows that the view coordination using highlights was a clear improvement to the current state of the art. It also shows that one of the most requested improvements was to show all comments in the comment view, regardless of the current comparison scope.

To sum up, the proposed solution is considered as improving the support to understanding the change for graphical model reviews. This was the main goal defined at the beginning of Section 3. Also all challenges mentioned in Section 1.1 have been met, so the proposed solution is considered as a success.

CHAPTER 7

Appendix

7.1 Case Examples

The subsequent sections give the provide the information that was given to the participants during the evaluation described in Section 5. A general information about the scenario is given first, followed by the descriptions of the individual tasks. The initial task descriptions are the problems that had to be solved by the change that is under review.

7.1.1 Event Management Background

You work for an event management company that organizes one-time events. Such events may span multiple days on one or more different event areas. Each event area is part of an event location and may provide a fixed number of seats. An event is organized in several categories, where each category may include other categories.

Customers buy tickets for a specific event and if seat reservation is required, also for specific seat. The ticket is not bound to the person that bought the ticket, so it is fine that one person buys multiple tickets for the same event.

The price of a ticket for an event is based on different price categories which also may differ from event to event. Not all of them require the reservation of seats and therefore seats are not necessarily assigned to price categories.

7.1.2 Event Management Models

The repository contains some Papyrus UML Models stored in the files *EventManagement.di*, *EventManagement.notation* and one or more *.uml*-files. The *.di* file contains data needed by the Papyrus editor and usually its content can be ignored. The *.uml* files contain the raw UML model, except diagram information like style, layout data etc.

Such information is stored in the diagram model which is located in the *.notation* file. During the interview, the following main UML Models can be found in the repository:

Class Diagram a class diagram that describes the data stored in our event management system.

Order Tickets An activity diagram that describes the process of ordering a ticket through the ticketing system after the user has selected an event.

Update Event Process An activity diagram that describes the process of updating the properties of an event with the event management system. This activity can be triggered manually or by an external event that requires someone to start actions of this activity. However, in both cases a user has to interact with the forms of this activity.

Event States A state diagram that contains the state(s) a event may have in the event management system.

A change may introduce new models or depend on another base version that may contain additional models.

7.1.3 Event Management Task “Event Cancellation”

Internal Gerrit Id: 26
Gerrit Change Id: I9575856e4d58a8bd4be66cb087400db9c5dc1133
Role: Reviewer

Initial Task Description Sometimes events get canceled due to unforeseen reasons. Such events have been simply deleted from the database until now. However, we also want to keep track of such events, so adapt the model accordingly. At least the date of cancellation and the reason for the cancellation must be stored.

7.1.4 Event Management Task “Event Subscription”

Internal Gerrit Id: 29
Gerrit Change Id: Ic368415b562368d9de807a2c36ecf1f7c43d8866
Role: Contributor

Initial Task Description We got the feedback that our customers want to be subscribe to some of the events and get notified of changes to the event. For now, adapt the model that customers may subscribe to an event when ordering tickets. Also include the notification of the subscribers in all diagrams if necessary.

7.1.5 Warehouse Background

You work for a company that manages a single warehouse which stores various products for customers. The products are delivered and stored on pallets which contain always only one type of a product. All of these pallets are stored on huge shelves inside the warehouse. Each of these shelves are divided into sections to facilitate the discovery of stored pallets.

Customers send the pallets through forwarders to store them for a specific amount of time until they are delivered back to the customer. This is done by trucks as the warehouse has no access to a railroad.

The company is legally bound to store the name, address and telephone number of each owner of a pallet stored in their warehouse. More detailed information can be found in the UML models in the repository.

7.1.6 Warehouse Models

The repository contains some Papyrus UML Models stored in the files *Warehouse.di*, *Warehouse.notation* and one or more *.uml*-files. The *.di* file contains data needed by the Papyrus editor and usually its content can be ignored. The *.uml* files contain the raw UML model, except diagram information like style, layout data etc. Such information is stored in the diagram model which is located in the *.notation* file. During the interview, the following main UML Models can be found in the repository:

Class Diagram a class diagram that describes the data stored in our warehouse system.

Pallet Unloading An activity diagram that describes the process of unloading pallet from a truck. This process is not limited to our warehouse system and also includes actions performed by any involved person.

Pallet Loading An activity diagram that describes the process of loading pallets for a specific delivery. This process is not limited to our warehouse system and also includes actions performed by any involved person.

Pallet States A state diagram that contains the state(s) a pallet may have in the warehouse system.

A change may introduce new models or depend on another base version that may contain additional models.

7.1.7 Warehouse Task “Damaged Pallets”

Internal Gerrit Id: 27
Gerrit Change Id: Ifa7e8593040e2fdbe45c3e8f3a2ed9d0cf31357b

Role: Reviewer

Sometimes, pallets get damaged by accidents in the warehouse, during the delivery, while loading, or while unloading. This has not been incorporated in the current model, so adapt the model accordingly. A pallet is considered as damaged if at least one product unit is damaged. It is also important to inform the owner of a pallet once a damaged pallet has been found.

7.1.8 Warehouse Task “Notifications”

Internal Gerrit Id: 30
Gerrit Change Id: I1cb04748660708f8dacd257c7f73c7c8663f06e6
Role: Contributor

Some of our customers requested that their contractors and customer get status notifications of their pallets. Adapt the model with the following notification system: A customer specifies a number of contacts for pallets which can be informed by SMS or Email. Each subscriber is notified once the pallets have been loaded or unloaded.

List of Figures

2.1 Screenshot: Gerrit Change Overview Page 10
2.2 Screenshot: Gerrit Difference Visualization 10
2.3 Common Review Process 22

3.1 Coordinated Views 32
3.2 Screenshot: Coordinated Views in Mervin 32
3.3 Screenshot: Unified Difference View in Mervin 35
3.4 Outline of the Graphical Element Structure 36
3.5 Graphical Element Structure Example 37
3.6 Graphical Element Structure Example Object Diagram 38
3.7 Illustration: Tab/Window Mode 39
3.8 Screenshots: Overlays 40
3.9 Illustration: Overlay Details 41
3.10 Illustration: Overlay Dependency 43
3.11 Illustration: Ambiguous Layout Outline 44
3.12 Illustration: Unification Problems without Layout Adjustment 45
3.13 Illustration: Off-Screen Indicators 47
3.14 Screenshot: Property Difference View in Mervin 49
3.15 Screenshot: Comment View in Mervin 51
3.16 Illustration: Comment Grouping 52
3.17 Screenshot: Version History View in Mervin 54
3.18 Screenshot: Review Explorer View in Mervin 57
3.19 Illustration: Difference Distribution Bars 58
3.20 Screenshot: Highlighting in Mervin 59

4.1 Simplified GEF architecture overview 65
4.2 Simplified GMF notation model 66
4.3 EMF Compare Architecture Excerpt 67
4.4 Mervin Review Model 69
4.5 Unified Notation Model Example 79
4.6 Illustration: Difference Similarity Example 88

5.1 Evaluation: Known Code Review Tools 98
5.2 Evaluation: Self-assessment Histograms 99

5.3 Evaluation: Missed Issue Error Rates 101
5.4 Evaluation: Participant Impressions 103

List of Tables

2.1 The version numbers of the analyzed code review tools 8

4.1 View Element Highlight Mapping 91
4.2 Highlight Element Selection Mapping 92

List of Algorithms

3.1 General comment grouping algorithm 53

4.1 Overview of the review (change) model loading process. 72
4.2 Overview of the patch loading process. 73
4.3 Overview of the model resource extraction process. 74
4.4 Overview of the comment loading process. 75
4.5 Overview of the comment data extraction process while saving comments. 75
4.6 Update process of the unified notation model 80
4.7 Unified model creation for a specific Papyrus diagram 81
4.8 Update references of copies created while merging deleted views 82
4.9 Update reference of a copy created while merging deleted views 83
4.10 Update container of a copy created while merging deleted views 84
4.11 Copying deleted views 85
4.12 Copy non-unifiable references (pseudo copies) 85

Index

Annotation, 21, 28, 29, 48, 50, 55, 61
  Color, 39, 49, 55
  Difference, 55
  Granularity, 21
  Overlay, 39
  Validity, 21, 29
Approval vote, 21
Artifact, 1, 58
  Annotation, 28, 29
  Base, 21, 60
  Changed, 60
  Difference type, 23
  Evolution, 29, 30
  Merger, 22
  Structure, 58
AttributeChange, 68
Author, 21
Automated tests, 8

Bar charts, 56
Bitbucket, 16

Case study, 93
Change
  Metrics, 28
  Outcome, 33
  Reason, 31, 33
  Set, 23
Code review tool, 7, 50
  Embedded, 8
  Standalone, 8
Command, 64
Comment, 21, 28, 40, 46, 50, 61
  Grouping, 50
  History, 50
  Link, 50, 61, 84
  Loading, 72
  Persistence, 71, 73
    Git Ref, 73
  Related, 50
  Replies, 50
  View, 33
Comparison, 49, 55, 61, 67
  Reference side, 33
  Resource filter, 67
  Scope, 31, 33, 37, 49, 55, 78
Component
  Focus viewport, 87
  Workbench, 86
    Containers, 86
Conflicting differences, 26
Coordinated views, 31
Copier, 82
Copy map, 82
Crucible, 12

Decision maker, 22
Deletion point, 49
Diagram, 24, 39, 60
  Readability, 46
Diff, 67, 68, 70
  Dependency, 68
Difference, 48, 53, 61
  Atomic operation, 34, 40, 46, 56, 61, 68, 76
    Addition, 34, 61
    Change, 34
    Deletion, 34, 61

Deletions, 55
Modification, 61
Cascading, 42
Computation
Framework, 35
Derived, 56
Evolution, 88
Image, 11, 26
Kind, 68
Layout, 34, 35, 40, 46, 56
Derivation, 34
Description, 56
Dimension, 34
Edge routing, 34
Glyphs, 42
Location, 34
Outline, 42, 43
Model mapping, 76
Modification, 34
Similarity, 53, 87, 89
Feature, 89
Kind, 89
Type, 89
Value, 89
Type, 23, 40, 49, 56
Differential, 18
Domain model, 2

Eclipse Modeling Framework, xi
Edit Policy, 65
Edition distance, 89
EditPart, 64
Evaluation
Case study
Audience, 97
Introduction, 97
Warm-up phase, 97
Error
Contributor task, 100
Incorrectly reported issue, 100
Missed issue, 98
Obvious unacceptable issue, 100
Recurring issue, 100
Reviewer task, 98
Unexpected issue, 100
Unreported missing requirement, 100
Feedback, 103
Improvements, 106
Issue
Expected, 94, 96
Justified, 95
Missed, 94
Obvious unacceptable, 96
Recurring, 95, 96
Reported unresolved, 96
Unexpected, 94, 96
Unjustified, 95
Unreported missing requirement, 96
Mervin, 96
Models, 97
Old tool set, 96
Participant workflow
Mervin, 102
Old tool set, 102
Prediction
Results, 105
Predictions, 94
Scenario, 96
Contributor, 97
Reviewer, 96
Self-assessment, 98
Subjective impressions, 103
User behavior, 100

File Evolution, 30
Future work
Comment grouping, 108
Comparison scope selection, 109
Deletion inconsistency problem, 108
Derived model elements for highlighting, 108
Further case studies, 109
Impact of explicit layout information, 108
Layout adjustment, 108
Metrics, 108

Navigation of highlighted graphical elements, 109
Reason of the change, 107
Related model elements, 108
Semantic model and notation model separation, 109

Gerrit, 1, 9
GitHub, 4, 15, 16, 98
GitLab, 17
GMF fork, 64
Graphical element, 35, 61
Anchor point, 35, 36, 77
Dimension, 36
Derived, 36, 45, 77
Edge, 35, 36, 45, 77
Node, 35
Containment, 43, 44
Nesting, 35
Position, 36
Absolute, 36
Relative, 36, 45, 77
Structure, 35
Graphical Model
Aspect, 39
Graphical representation, 35, 48, 61
Graphical syntax, 2, 24, 35

Highlighting, 50, 59, 90
Associated elements, 91
Clearing, 59
Considerations, 59
Derived, 60
Graphical elements, 60
Label, 60
Navigation, 60
Permanent, 59
Request, 60
Selection, 90, 91
Service, 91, 92
Temporary, 59

Input data set, 60
Insertion point, 49

Kallithea, 20

Layer, 65, 78, 86
Connection, 66
Layout
Adjustment, 37, 44, 45
Constraint, 2, 45, 66, 77
Mapping, 76
Information, 34
Conversion, 34
Explicit, 34
Implicit, 34
Semantic, 45
Manager, 65
Lazy loading, 71

Match, 61, 67
Matcher, 68
Matching, 89
Merge, 63
Conflict, 26, 68
Request, 23
State, 26
Mervin repository, 63
Metamodel, 24
Model, 24, 48, 50
Element, 61
Contained differences, 49
Context, 48, 49
Hint, 78
Metric, 56
Structure, 48, 55
Types, 58
Modeling language, 2
Move direction
Mapping, 76

Navigation, 46, 59
Highlighting, 60
Notation model
Anchor, 66
Bendpoint, 66
Diagram, 65, 66
Edge, 65

Node, 65
View, 65

Off-screen indicator
Location, 47
Merged, 48
Merger, 87
Projection, 47
Shadow, 48
Orthographic projection, 47
Overlay, 39, 61
Color, 40
Creation, 84
Dependency, 42, 86
Model, 43
Encoding, 40
Filter, 42, 43, 46, 87
Interaction, 42
Semi-transparency, 42
Visibility, 42

Papyrus, xi
Patch, 58
Loading, 71
Metadata, 71
Set
Loading, 72
Reference, 73
Type, 71
Patchwork, 13
Property difference view, 33, 64
Pseudo copy, 80–82
Creation, 84
Overlay, 86
Pull request, 23

Radial projection, 47
ReferenceChange, 68
Rejection vote, 21
Remote repository, 71
Resource set cache, 71, 74
Review
Challenges, 23
Data, 61, 63
Explorer view, 33, 55, 64
Model, 68, 71, 78
Comment, 70
DiagramResource, 70
Difference, 70
DifferenceOverlay, 68, 70
EObject, 70
LayoutDifference, 70
ModelResource, 70
ModelReview, 68
Patch, 70
PatchSet, 68
StateDifference, 70
Process, 1, 7, 21, 29, 50
Request, 31
Requester, 21
Roles, 8, 21, 22
ReviewBoard, 11, 28
Reviewer, 21, 50
RhodeCode, 19
Rietveld, 9

Selection, 48
Side by side view, 23, 26, 27, 49
StateDifferenceType, 76
Structural Feature, 68

Task, 21
Textual syntax, 2
Tool set, 7, 93
Tool support factors, 93
Identification of differences, 94
Identification of issues, 94
Identification of linked elements, 94
Navigation through linked elements, 94
Tracking of differences, 94
Tracking of issue discussions, 94
Verification of reported issues, 94

Understand, 14
Unified diff format, 9
Unified difference view, 33, 35, 77
Tab mode, 39

Window mode, 39
Unified model, 78, 79
Creation, 81
Unified view, 23
Upsource, 13
URI
Converter, 72
Handler, 73

Version, 23, 31, 34, 48–50, 52, 55, 60
Base, 55
History view, 33, 52, 87, 91
Container, 53, 90
Filter, 55
Left, 31
New, 31, 37, 55
Old, 31, 37
Right, 31
View
Component, 65
Coordination, 59
Layout, 33
Parent hierarchy, 86
Scalability, 33
Visualization
Difference type
Distribution, 56
Annotations, 2, 50
Comparative, 25
Difference type
Indicators, 56
Non-model artifacts, 58
Review history, 3, 52, 87
Scalability, 2
Side by side, 25, 27, 48
Unified, 25, 27, 28

Watcher, 22
Window to the past, 46

XYLayout, 77, 80

Glossary

Android Open Source Project The developers of the well known Android Operating system released the Android Stack as an open source project, which is named the Android Open Source Project [Pro17]. 1

Change A change is the main entity that is under review in a review process. It contains multiple sets of changed artifacts that are applied to a set of existing artifacts to solve a certain task. More information can be found in Section 2.2. 10, 21, 68, 71–74, 109, 119, 129

Code Review A process used in software development to find and prevent bugs by approval of an proposed source code change through additional persons or automated process. xi, xiii, 1, 2, 4, 7–9, 11–14, 17–19, 25, 26, 33, 34, 107, 117

Eclipse Eclipse is mostly known for the Eclipse IDE, but is actually an open source community that keeps the development of various projects running [Ecl17a]. 1, 4, 63

Eclipse Modeling Framework A framework that supports the creation and development of models with java [Gro17]. xi, xiii, 4, 26, 131

Eclipse Platform A set of frameworks that form the base infrastructure for eclipse rich client software. 63, 93, 128, 129

Eclipse Public Licence An open source license provided by the eclipse foundation which can be found at `https://www.eclipse.org/legal/epl-v10.html`. 63, 131

EGit EGit provides Git support for the Eclipse IDE. More information can be found at `http://www.eclipse.org/egit/`. 93, 96

EMF Compare A framework that provides support for comparing EMF models [Ecl17c]. 26, 34, 35, 42, 48, 55, 63, 64, 66–68, 70, 72, 74, 76, 82, 87, 89, 93, 96, 100, 109

Gerrit A code review tool based on Git [Goo17b]. For more information see Section 2.1.1. 1, 9–11, 63, 68, 71–73, 93, 96, 107

Git A decentralized version control system. See `https://git-scm.com/` for more details. 64, 71–73, 127, 129

Graphical Editing Framework An eclipse framework that provides facilities to create and develop graphical editors using the Eclipse Platform [Nys17]. 63, 131

Graphical Model Models described by a graphical syntax. 4, 24–26, 30, 33, 35–37, 44–46, 56, 60, 78, 93, 96, 106–108, 128

Graphical Model Review The process of reviewing diagrams or models with a graphical syntax. 7, 8, 23, 25, 31, 34, 46, 106–110

Graphical Modeling Framework An eclipse framework that extends GEF with EMF to create and develop graphical editors for graphical models [Ecl17e]. 24, 131

LibreOffice An open source application suite for office applications [Fou17]. 1

Maven A build tool that is used to build the prototype described in this thesis. More information can be found at `https://maven.apache.org/`. 63

Mental Map Each person creates an abstract structure of layouts in their minds when they view graphs. This abstract structure is called the Mental map [DG02, ELMS91, MELS95]. 2, 28

Mervin Mervin is the name of the prototype that has been developed to evaluate the proposed solution in this thesis. For more information see Section 4. 5, 31, 32, 35, 40, 49, 51, 59, 63–69, 71, 74, 76–82, 84, 86, 87, 89–93, 100, 102–107, 109

Model Evolution The evolution of changed models in Model versioning [BKL+12]. 4

Model Review The process of reviewing models with any syntax. 107

Model Versioning Model Versioning is the extension of the idea of the well known source code or text based versioning approach on models [BKL+12]. 4, 128

Model-Driven Engineering Model-Driven Engineering is a paradigm that uses models as the main artifacts in development to generate executables [BKL+12]. 2, 4, 24, 97–99

Mylyn Mylyn is a framework for eclipse that provides support for managing development tasks including code reviews. More information can be found at `http://www.eclipse.org/mylyn/`. 93, 96, 100, 103

Notation Model The notation model is a model defined by GMF that is used to separate the graphical description of model elements from the semantic model. It describes the graphical elements with their properties that are not contained in the semantic model or cannot be derived from thesemantic model. 24, 65, 66, 74, 76–81, 84, 91, 92, 119, 129

Off-screen Indicator Indicators attached to the sides of a view to show elements not presented in the current screen or view area. 46, 47, 78, 86, 87

Papyrus A modeling environment for various standardized modeling languages including UML, based on the Eclipse Platform. xi, xiii, 4, 26, 27, 35, 55, 58, 63, 64, 66, 76, 78–81, 86, 87, 90, 93, 96–100, 102, 103, 105, 107–109, 119

Review History Changes contain various versions of changed artifacts that are created at different times. These artifacts form a history of changed artifacts, which is called the Review History in this thesis. 1

Screen Space The amount of space that can be used to show the elements on one or more combined display devices. 33

Semantic Model The actual EMF model that is referenced by the notation model in GMF. 24, 65, 66, 76, 84, 91, 92, 128

Unified Model Map A map that links the copied elements in the original notation model with their copies in the unified notation model. It also stores the pseudo copies, the elements that needed to be copied during unification due to constraints of the notation model. 68, 79–82, 84, 91

Version Control System A software that allows developers to create, store and relate different versions of artifacts. Git is an example for such a system. 8, 131

Visual Information Seeking Mantra This mantra has been defined by Ben Shneiderman and contains visual design guidelines important for information visualization applications [Shn96]. It reads as follows: Overview first, zoom and filter, then details-on-demand [Shn96]. 2, 42, 48, 56

Acronyms

API Application Programming Interface. 13, 26, 71, 78

CVD color vision deficiency. 23

EMF Eclipse Modeling Framework. 26, 48, 63, 65, 66, 71–73, 82, 90, 127–129, *Glossary:* Eclipse Modeling Framework

EPL Eclipse Public Licence. 63, *Glossary:* Eclipse Public Licence

GEF Graphical Editing Framework. 63–66, 86, 128, *Glossary:* Graphical Editing Framework

GMF Graphical Modeling Framework. 24, 26, 54, 55, 57, 58, 63–66, 68, 70, 77, 78, 80, 81, 86, 107, 128, 129, *Glossary:* Graphical Modeling Framework

IDE Integrated Development Environment. 8

MVC Model View Controller. 64

REST Representational State Transfer. 13

UML Unified Modeling Language. xi, xiii, 45, 63, 66, 78, 93, 96–100, 107, 109, 129

URI Uniform Resource Identifier. 72–74

VCS Version Control System. 8, 11–13, 15, 16, 18, 19, 70, *Glossary:* version control system

Bibliography

[Atl17a] Atlassian. Bitbucket | the git solution for professional teams. `https://bitbucket.org/product`, 2017. Accessed: 2017-05-17.

[Atl17b] Atlassian. Crucible - code review tool for SVN, git, perforce and more. `https://www.atlassian.com/software/crucible/overview`, 2017. Accessed: 2017-03-29.

[BB13] Alberto Bacchelli and Christian Bird. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 712–721, Piscataway, NJ, USA, 2013. IEEE Press.

[BE94] Marla J. Baker and Stephen G. Eick. Visualizing software systems. In *Proceedings - International Conference on Software Engineering*, pages 59–67. Publ by IEEE, 1994.

[BE96] T. Ball and S. G. Eick. Software visualization in the large. *Computer*, 29(4):33–43, 1996.

[Bea17] Beanbag, Inc. Take the pain out of code review | review board. `https://www.reviewboard.org/`, 2017. Accessed: 2017-03-29.

[BKL+12] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. An introduction to model versioning. In *Formal Methods for Model-Driven Engineering*, pages 336–398. Springer, 2012.

[BR07] E. Baudrier and A. Riffaud. A Method for Image Local-Difference Visualization. In *Document Analysis and Recognition, 2007. ICDAR 2007. Ninth International Conference on*, volume 2, pages 949–953. IEEE, 2007.

[Bé05] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, May 2005.

[CAT07] Fanny Chevalier, David Auber, and Alexandru Telea. Structural Analysis and Visualization of C++ Code Evolution Using Syntax Trees. In *Ninth International Workshop on Principles of Software Evolution: In Conjunction*

with the 6th ESEC/FSE Joint Meeting, IWPSE '07, pages 90–97, New York, NY, USA, 2007. ACM.

[CKN+03] Christian Collberg, Stephen Kobourov, Jasvir Nagra, Jacob Pitts, and Kevin Wampler. A System for Graph-based Visualization of the Evolution of Software. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, SoftVis '03, pages 77–ff, New York, NY, USA, 2003. ACM.

[DC03] Gareth Daniel and Min Chen. Video Visualization. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pages 54–, Washington, DC, USA, 2003. IEEE Computer Society.

[DG02] Stephan Diehl and Carsten Görg. Graphs, they are changing. In *Graph drawing*, pages 23–31. Springer, 2002.

[Ecl17a] Eclipse Foundation, Inc. Eclipse. https://eclipse.org/home/index.php, 2017. Accessed: 2017-03-29.

[Ecl17b] Eclipse Foundation, Inc. Eclipse Public License - Version 1.0. https://www.eclipse.org/org/documents/epl-v10.html, 2017. Accessed: 2017-04-18.

[Ecl17c] Eclipse Foundation, Inc. EMF compare - compare and merge your EMF models. https://www.eclipse.org/emf/compare/, 2017. Accessed: 2017-03-29.

[Ecl17d] Eclipse Foundation, Inc. EMF compare - developer guide. https://www.eclipse.org/emf/compare/documentation/latest/developer/developer-guide.html, 2017. Accessed: 2017-03-29.

[Ecl17e] Eclipse Foundation, Inc. GMF. https://www.eclipse.org/modeling/gmp/, 2017. Accessed: 2017-03-29.

[Ecl17f] Eclipse Foundation, Inc. Papyrus. https://eclipse.org/papyrus/, 2017. Accessed: 2017-04-18.

[ELMS91] Peter Eades, Wei Lai, Kazuo Misue, and Kozo Sugiyama. *Preserving the mental map of a diagram.* International Institute for Advanced Study of Social Information Science, Fujitsu Limited, 1991.

[FG04] Michael Fischer and Harald Gall. Visualizing feature evolution of large-scale software based on problem and modification report data. 2004.

[Fou17] The Document Foundation. Home | LibreOffice - free office suite - fun project - fantastic people. https://www.libreoffice.org/, 2017. Accessed: 2017-03-29.

[Fre17] Free Software Foundation, Inc. Diffutils manual - diff output formats. https://www.gnu.org/software/diffutils/manual/diffutils.html#Output-Formats, 2017. Accessed: 2017-05-17.

[GAW+11] Michael Gleicher, Danielle Albers, Rick Walker, Ilir Jusufi, Charles D. Hansen, and Jonathan C. Roberts. Visual comparison for information visualization. *Information Visualization*, 10(4):289–309, October 2011.

[Git17a] GitHub, Inc. GitHub - Build software better, together. https://github.com, 2017. Accessed: 2017-05-17.

[Git17b] GitLab B.V. Code, test, and deploy together with gitlab open source git repo management software. https://about.gitlab.com/, 2017. Accessed: 2017-05-17.

[GK10] Martin Graham and Jessie Kennedy. A Survey of Multiple Tree Visualisation. *Information Visualization*, 9(4):235–252, 2010.

[Goo17a] Google Inc. Gerrit code review - gerrit installations in the wild. https://gerrit.googlesource.com/homepage/+/md-pages/docs/ShowCases.md, 2017. Accessed: 2017-03-29.

[Goo17b] Google Inc. Gerrit code review - index.md. https://www.gerritcodereview.com/, 2017. Accessed: 2017-03-29.

[Gro17] Richard Gronback. Eclipse modeling project. https://eclipse.org/modeling/emf/, 2017. Accessed: 2017-03-29.

[GS03] Jack Greenfield and Keith Short. Software factories: assembling applications with patterns, models, frameworks and tools. pages 16–27. ACM, October 2003.

[HMPR04] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004.

[Jet17] JetBrains s.r.o. Upsource: Polyglot code review tool. https://www.jetbrains.com/upsource/, 2017. Accessed: 2017-05-13.

[JH06] Luke Jefferson and Richard Harvey. Accommodating Color Blind Computer Users. In *Proceedings of the 8th International ACM SIGACCESS Conference on Computers and Accessibility*, Assets '06, pages 40–47, New York, NY, USA, 2006. ACM.

[Ker17] Jeremy Kerr. Patchwork. http://jk.ozlabs.org/projects/patchwork/, 2017. Accessed: 2017-05-10.

[Lan01] Michele Lanza. The evolution matrix: recovering software evolution using software visualization techniques. pages 37–42. ACM, September 2001.

[Lee89] Allen S. Lee. A Scientific Methodology for MIS Case Studies. *MIS Quarterly*, 13:33–50, 1989.

[LGJ] Yuehua Lin, Jeff Gray, and Frédéric Jouault. DSMDiff: a differentiation tool for domain-specific models. 16(4):349–361.

[MELS95] Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama. Layout adjustment and the mental map. *Journal of visual languages and computing*, 6(2):183–210, 1995.

[MGH] Akhil Mehra, John Grundy, and John Hosking. A generic approach to supporting diagram differencing and merging for collaborative design. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 204–213. ACM.

[NS00] Chris North and Ben Shneiderman. Snap-together visualization: can users construct and operate coordinated visualizations? *International Journal of Human-Computer Studies*, 53(5):715–739, November 2000.

[Nys17] Alexander Nyssen. GEF. `https://eclipse.org/gef/`, 2017. Accessed: 2017-03-29.

[OWKa] Dirk Ohst, Michael Welle, and Udo Kelter. Difference tools for analysis and design documents. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings*, pages 13–22. IEEE.

[OWKb] Dirk Ohst, Michael Welle, and Udo Kelter. Differences between versions of UML diagrams. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, pages 227–236. ACM.

[Pha17] Phacility, Inc. Phacility - products - differential. `https://www.phacility.com/phabricator/differential/`, 2017. Accessed: 2017-05-17.

[Pro17] Android Open Source Project. Android open source project. `https://source.android.com/`, 2017. Accessed: 2017-03-29.

[Rho17] RhodeCode, Inc. Enterprise code management for hg, Git, SVN. `https://rhodecode.com/`, 2017. Accessed: 2017-03-29.

[Sch06] D.C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, February 2006.

[Sci17] Scientific Toolworks, Inc. Code review tool. `https://scitools.com/code-review/`, 2017. Accessed: 2017-05-9.

[Sel03] B. Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, September 2003.

[SFvH] A. Schipper, H. Fuhrmann, and R. von Hanxleden. Visual comparison of graphical models. In *2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, pages 335–340. IEEE.

[Shn96] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 336–343. IEEE, 1996.

[Sim10] M. P. Simunovic. Colour vision deficiency. *Eye*, 24(5):747–755, May 2010.

[Sof17] Software Freedom Conservancy. Kallithea. `https://kallithea-scm.org/`, 2017. Accessed: 2017-05-17.

[SR04] E. Suvanaphen and J. C. Roberts. Textual difference visualization of multiple search results utilizing detail in context. In *Proceedings Theory and Practice of Computer Graphics, 2004.*, pages 2–8, June 2004.

[Stö] Harald Störrle. Making sense of UML class model changes by textual difference presentation. In *Proceedings of the 6th International Workshop on Models and Evolution*, ME '12, pages 3–8. ACM.

[TA08] Alexandru Telea and David Auber. Code Flows: Visualizing Structural Evolution of Source Code. *Computer Graphics Forum*, 27(3):831–838, 2008.

[vdBPV] Mark van den Brand, Zvezdan Protić, and Tom Verhoeff. Generic tool for visualization of model differences. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, IWMCP '10, pages 66–75. ACM.

[vR17] Guido van Rossum. Code review, hosted on google app engine. `https://github.com/rietveld-codereview/rietveld`, 2017. Accessed: 2017-05-9.

[VTvW05] Lucian Voinea, Alex Telea, and Jarke J. van Wijk. CVSscan: visualization of code evolution. pages 47–56. ACM, May 2005.

[Wen] Sven Wenzel. Scalable visualization of model differences. In *Proceedings of the 2008 International Workshop on Comparison and Versioning of Software Models*, CVSM '08, pages 41–46. ACM.

[ZMG+03] Polle T. Zellweger, Jock D. Mackinlay, Lance Good, Mark Stefik, and Patrick Baudisch. City lights: Contextual views in minimal space. In *CHI '03 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '03, pages 838–839. ACM, 2003.